CAPITULO 1

ARQUITECTURAS

1.1. - INTRODUCCIÓN

El tema de Arquitecturas aquí tratado no se enfoca desde el punto de vista de la ingeniería electrónica, sino desde el punto de vista funcional, ya que nuestra meta es llegar a mostrar qué componentes existen y cómo pueden ser amalgamados para cumplir una función específica más que analizar su estructura real.

Salvo casos expresamente identificados, el tema de Sistemas Operativos se referirá siempre a casos multitarea multiusuarios.

Realizaremos en esta primera parte todo un pantallazo sobre los temas de arquitecturas y sistemas operativos a efectos de homogeneizar conceptos y terminología.

1.2. - SISTEMAS OPERATIVOS

En esta primera parte trataremos de responder a preguntas del tipo : Qué es un Sistema Operativo, qué hace, y para qué sirve ?. De esta forma queremos mostrar dónde queremos llegar para ver posteriormente la forma en que lograremos esto en capítulos subsiguientes.

Las dos funciones primordiales de un S.O. son las siguientes :

1)- **La utilización compartida de recursos**

2)- **La constitución de una máquina virtual**

En el primer punto es importante saber que un sistema operativo debe lograr que se compartan los recursos de un computador entre un cierto número de usuarios que trabajan en forma simultánea (obviamente en sistemas multiusuario ó multitarea). Esto es importante ya que de esta forma se busca incrementar la disponibilidad del computador con respecto a esos usuarios y, al mismo tiempo, maximizar la utilización de recursos tales como el procesador central, la memoria y los periféricos de E/S.

La segunda función de un Sistema Operativo es la de transformar un cierto hardware en una máquina que sea fácil de usar. Esto es equivalente a colocar frente al usuario una **máquina virtual** cuyas características sean distintas y más fáciles de manejar que la máquina real subyacente.

Un ejemplo concreto y muy conocido en el cual la máquina virtual difiere de la real es por ejemplo en el campo de las E/S, ya que en este caso el hardware básico puede que sea extremadamente difícil de manejar por el usuario y que requiera de sofisticados programas. Nuestro Sistema Operativo deberá entonces evitar al usuario el problema de tener que comprender el funcionamiento de este hardware poniendo a su alcance una máquina virtual mucho más sencilla pero con las mismas posibilidades de E/S.

La naturaleza de una máquina virtual dependerá de la aplicación específica para la cual se la quiera utilizar. Evidentemente el diseño del Sistema Operativo de esta máquina estará fuertemente influenciado por el tipo de aplicación que se quiera dar a la máquina.

Existen máquinas de **propósito general** en las cuales no se puede identificar un tipo de aplicación concreta. Tales máquinas son muy criticadas en el sentido de que por tratar de ofrecer prestaciones en un espectro muy general de aplicaciones no pueden efectivamente responder eficientemente en ninguna de ellas.

1.2.1. - Funciones de un Sistema Operativo

Los sistema operativos deben llevar a cabo como mínimo las siguientes funciones :

- Secuenciar las tareas : llevar un cierto orden respecto de los diferentes trabajos que debe realizar.
- Interpretar un lenguaje de control : debe comprender los diferentes tipos de órdenes que se le imparten para poder ejecutar las tareas.
- Administrar errores : debe tomar las adecuadas acciones según los diferentes tipos de errores que se produzcan como así también el permitir la intervención externa de, por ejemplo, el operador.
- Administrar las interrupciones : debe interpretar y satisfacer todas las interrupciones que se puedan llegar a producir.
- Scheduling : administrar equitativamente el recurso procesador entre las diferentes tareas.
- Controlar los recursos existentes : debe llevar debida cuenta de todos los recursos como así también del estado en que se encuentran y las funciones que realizan en todo momento.
- Proteger : debe proteger la información de todos los usuarios entre sí, como así también debe asegurar la integridad de los datos.
- Debe permitir el procesamiento interactivo.
- Debe ser de fácil interacción para los usuarios.
- Debe llevar un control global de todos los recursos del sistema.

En virtud de esta enumeración podemos extraer de la lista anterior una serie de características que debe poseer un Sistema Operativo.

1.2.2. - **Características de un Sistema Operativo**

1.2.2.1. - **Concurrencia**

La concurrencia consiste en la existencia de varias actividades simultáneas o paralelas. Por ejemplo la superposición de las E/S con la ejecución del cómputo.

La concurrencia lleva asociado los siguientes problemas :
- Conmutar de una tarea a otra.
- Proteger una determinada actividad de los efectos de otra
- Sincronizar las tareas que sean mutuamente dependientes.

Definiremos el procesamiento en paralelo como :

Definición *: El procesamiento paralelo es una eficiente forma de procesar la información que pone énfasis en la explotación de eventos concurrentes en el proceso de cómputo. La concurrencia implica* ***paralelismo****,* ***simultaneidad*** *y* ***pipelining****.*

Los *eventos paralelos* pueden ocurrir en diferentes recursos en el mismo intervalo de tiempo, la multiprogramación es un ejemplo clásico de paralelismo ya que dado un intervalo de tiempo la CPU, vista desde la óptica del usuario, simula que todos los programas son ejecutados al mismo tiempo.

Los *eventos simultáneos* pueden ocurrir en el mismo instante del tiempo, por ejemplo un sistema que cuente con dos CPU's estaría ejecutando simultáneamente instrucciones en cada una de ellas.

Los *eventos pipeline* pueden ocurrir en intervalos de tiempo superpuestos, por ejemplo si se tuviera una unidad hardware interna a la CPU que interpretase el código de operación de una instrucción y otra unidad que cargase de memoria la próxima instrucción a ejecutar, mientras se decodifica el código de la instrucción actual se estaría cargando la próxima instrucción en la CPU, luego se superponen en el tiempo ambas funciones.

Estos eventos concurrentes se alcanzan en un sistema de computación con varios niveles de procesamiento. El procesamiento paralelo necesita de la ejecución concurrente de muchos programas en el computador. Esto es justamente la contrapartida del procesamiento secuencial. En suma, esta es una forma altamente eficiente desde el punto de vista del costo, de mejorar la performance de un sistema mediante actividades concurrentes en el computador.

1.2.2.2. - **Utilización conjunta de recursos**

Puede ser que varias actividades concurrentes tengan que compartir determinados recursos o información. Hay cuatro razones para ello :
- el costo; es absurdo disponer de infinitos recursos.
- la posibilidad de trabajar a partir de lo que hicieron otros.
- la posibilidad de compartir datos.
- la eliminación de información redundante.

Asociado a esto tenemos los problemas de ubicar y proteger estos recursos, el acceso simultáneo a la información y la ejecución simultánea de programas.

1.2.2.3. - **Almacenamiento a largo plazo**

Este tipo de almacenamiento permite que el usuario guarde sus datos o programas en el propio computador.

El problema aquí es el de proporcionar un acceso fácil a estos datos, la protección de la información y el resguardo de la misma ante fallas del sistema.

1.2.2.4. - **Indeterminismo**

Un Sistema Operativo debe ser determinista en el sentido de que el mismo programa ejecutado con los mismos datos ayer u hoy debe producir los mismos resultados. Por otro lado es indeterminista en el sentido de que debe responder a circunstancias que pueden ocurrir en un orden impredecible.

Debido a la gran cantidad de eventualidades que pueden darse, es evidentemente poco razonable esperar poder escribir un sistema operativo que las considere todas una a una. En lugar de ello, el sistema debe escribirse de forma que sea capaz de tratar cualquier secuencia de circunstancias de este tipo.

1.2.3. - **Características Deseables**

1.2.3.1. - **Eficiencia**

Si bien la eficiencia de un S.O. es muy importante lamentablemente es difícil establecer un criterio único por el cual pueda juzgarse. Algunos criterios posibles son :

- Tiempo transcurrido entre tareas
- Tiempo ocioso del procesador central
- Tiempo de ejecución de las tareas batch
- Tiempo de respuesta en los sistemas interactivos
- Utilización de los recursos
- Rendimiento (tareas ejecutadas por hora)

No todos estos criterios pueden satisfacerse simultáneamente, siempre que se privilegia uno de ellos es en detrimento de algún otro.

1.2.3.2. - **Fiabilidad**

Teóricamente un Sistema Operativo debe estar completamente libre de todo tipo de errores y ser capaz de resolver satisfactoriamente todas las contingencias que pudiesen presentársele. En la práctica ello nunca ocurre, aunque en este punto se han logrado notables avances.

1.2.3.3. - **Facilidad de corrección**

Debería ser posible corregir o mejorar un S.O. sin tener que hacer uso de todo un ejército de programadores. Esto se puede lograr si el sistema es de construcción modular con interfases claramente definidas entre los diferentes módulos.

La idea aquí es lo que se suele denominar **Sistema Abierto**, al cual, por ejemplo es fácil agregarle drivers (programas especiales para manejar ciertas situaciones o periféricos) que se toman en el momento de la carga o porque sus módulos son sencillos de linkeditar en un único código objeto.

1.2.3.4. - **Tamaño pequeño**

El espacio que consume en la memoria el sistema operativo debería esperarse que fuera pequeño ya que cuanto mayor es el mismo provoca que exista una zona mayor no destinada a tareas de los usuarios. Además un sistema grande está más sujeto a errores

1.2.4. - **PROCESOS CONCURRENTES**

Antes de iniciar un estudio más detallado de los sistemas operativos, introduciremos algunos conceptos fundamentales.

1.2.4.1. - **Programas, Procesos y Procesadores**

Consideremos un sistema operativo como un conjunto de actividades cada una de las cuales lleva a cabo una cierta función, como por ejemplo la administración de las E/S. Cada una de estas actividades consiste en la ejecución de uno o más programas que se ejecutarán toda vez que se requiera tal función.

Utilizaremos la palabra **proceso** para referirnos a una actividad de este tipo.

Consideremos a un programa como una entidad pasiva y a un proceso como una entidad activa, un **proceso** consiste entonces, en una secuencia de acciones llevadas a cabo a través de la ejecución de una serie de instrucciones (un programa), cuyo resultado consiste en proveer alguna función del sistema. Las funciones del usuario también pueden asimilarse a este concepto, es decir que la ejecución de un programa de un usuario también será un proceso.

Un proceso puede involucrar la ejecución de más de un programa. Recíprocamente, un determinado programa o rutina pueden estar involucrados en más de un proceso. De ahí que el conocimiento del programa en particular que está siendo ejecutado no nos diga mucho acerca de la actividad que se está llevando a cabo o de la función que está siendo implementada. Es fundamentalmente por esta razón por lo que es más útil el concepto de proceso que el de programa.

Un proceso es llevado a cabo por acción de un agente (unidad funcional) que ejecuta el programa asociado. Se conoce a esta unidad funcional con el nombre de **procesador**.

Los conceptos de proceso y de procesador pueden emplearse con el fin de interpretar tanto la idea de concurrencia como la de no-determinismo. La concurrencia puede verse como la activación de varios procesos a la vez. Suponiendo que haya tantos procesadores como procesos esto no reviste inconveniente alguno. Pero, si sucede como habitualmente que los procesadores son menos que los procesos se puede lograr una concurrencia aparente conmutando los procesadores de uno a otro proceso. Si esta conmutación se lleva a cabo en intervalos lo suficientemente pequeños, el sistema aparentará un comportamiento concurrente al ser analizado desde la perspectiva de una escala mayor de tiempo.

Resumiendo, un proceso es una secuencia de acciones y es, en consecuencia, dinámico, mientras que un programa es una secuencia de instrucciones y es, así pues, estático. Un procesador es el agente que lleva a cabo un proceso. El no-determinismo y la concurrencia pueden describirse en términos de interrupciones de procesos entre acciones (imposibilidad de prever el momento en el que se produce la interrupción) y de conmutación de procesadores entre procesos (se reparte el tiempo de CPU en rebanadas asignadas a cada proceso). Con el fin de que pueda llevarse a cabo esta conmutación, debe guardarse suficiente información acerca del proceso con el fin de que pueda reemprenderse más tarde.

1.2.4.2. - **Comunicación entre procesos**

Los distintos procesos dentro de un computador no actúan, evidentemente, de forma aislada. Por un lado deben cooperar con el fin de alcanzar el objetivo de poder ejecutar las tareas de los usuarios, y compiten por el uso de los diferentes recursos como ser memoria, procesador y archivos.
Estas dos actividades llevan asociada la necesidad de algún tipo de comunicación. Las áreas en las que esta comunicación es esencial pueden dividirse en :

1.2.4.2.1. - **Exclusión Mutua**

Existen recursos en un sistema de tipo compartibles (es decir pueden ser usados por varios procesos en forma concurrente) o no-compartibles (o sea que su uso se ve restringido a un solo proceso por vez). El hecho de no ser compartible puede deberse a cuestiones de imposibilidad técnica o por la razón de que es necesario su uso individual en virtud de que algún otro proceso pueda interferir con el que lo está usando.
El problema que trata la exclusión mutua es asegurar que los recursos no-compartibles sean accedidos por un solo proceso a la vez.

1.2.4.2.2. - **Sincronización**

En términos generales la velocidad de un proceso respecto a otro es impredecible ya que depende de la frecuencia de la interrupción asociada a cada uno de ellos y de cuán a menudo y por cuánto tiempo tuvo asignado el recurso procesador. Se dice entonces, que un proceso se ejecuta **asincrónicamente** respecto de otro.
Sin embargo existen ciertos instantes en los cuales los procesos deben sincronizar sus actividades. Son estos puntos a partir de los cuales un proceso no puede continuar hasta tanto se haya completado algún tipo de actividad. El Sistema Operativo debe proveer mecanismos que permitan poder llevar a cabo esta sincronización.

1.2.4.2.3. - **Deadlock - Abrazo Mortal**

Cuando varios procesos compiten por los recursos es posible que se de una situación en la cual ninguno de ellos puede proseguir debido a que los recursos que necesita están ocupados por otro proceso. Esta situación se conoce con el nombre de **deadlock**. El evitarlos o al menos limitar sus efectos, es claramente una de las funciones de los sistemas operativos.

Hasta aquí hemos tratado de dar un panorama general sobre lo que son los sistemas operativos. Entrando más de lleno en el temario pasamos ahora a desarrollar los conceptos de arquitecturas.

1.3. - **ARQUITECTURA DE COMPUTADORES**

1.3.1. - **Algunas Definiciones**

Veamos cuales son algunas de las definiciones respecto de sobre qué es la arquitectura de un computador :

- Arte de diseñar una máquina con la cual sea agradable trabajar ". (Caxton Foster - 1970)
- Determinar componentes, funciones de los componentes y reglas de interacción entre ellos " (N. Prassard - 1981)
- La estructura de la Computadora que el programador necesita conocer con el objeto de escribir programas en lenguaje de máquina correctos " (Informe final del proyecto CFA (Arquitectura de Familias de Computadoras).

De acuerdo a esto, cada cual define la Arquitectura de un computador, como ya es obvio, desde su punto de vista e interés de estudio.
La última definición hace hincapié en la interfase que el hardware le presenta al programador.
La segunda hace referencia a la estructura, organización, implementación y comportamiento del computador.

Es evidente que ninguna de las tres definiciones por sí solas dan un entendimiento cabal de lo que es una Arquitectura y aún aceptando que ambas se complementen, faltan aspectos de algoritmos y estructuras lógicas para completar la definición, como el que correspondería a como se ejecuta en realidad una instrucción.

Ténganse en cuenta que :

- Estructura : hace referencia a la interconexión entre los distintos componentes. Por ejemplo que el diseño tenga una CPU, una memoria y un canal de E/S y que la CPU se conecte entre la memoria y el canal de E/S.
- Organización : hace referencia a las interacciones dinámicas y de administración de los componentes. Siguiendo el ejemplo anterior la CPU actuará como comunicación entre las E/S que se realicen cargando los datos transferidos desde el canal a la memoria, asimismo atenderá las señales que se envíen desde el canal y enviará comandos al mismo.
- Implementación : hace referencia al diseño específico de los componentes. Por ejemplo puede ser el tipo de acceso a memoria en cuanto a si es por dirección o por contenido, para lo cual el hardware necesario difiere sensiblemente.

Es válido recordar que la implementación distingue dos computadoras distintas pero de igual arquitectura (Ejemplo : Arquitectura secuencial con o sin Pipeline; en el primer caso la arquitectura requiere de un hardware específico que es el pipeline).

En definitiva una arquitectura está dada por sus :

- Componentes
- Interconexión de sus componentes
- Interacción entre sus componentes
- Implementación de sus componentes
- La forma de usar todos estos aspectos en beneficio de una tarea o sea su **operación** (qué y cómo hace cada instrucción) aliviando o **no** al software en su tarea (microcódigos, subsistema de E/S, etc.).

1.3.2. - NIVELES DE ARQUITECTURAS

Teniendo en cuenta los puntos anteriores se puede hablar de niveles de arquitectura :

- **Exoarquitectura** : Es la estructura y capacidad funcional de la arquitectura visible al programador de assembler (qué y cómo hace cada instrucción, por ejemplo una instrucción MVCL -move character long- se utiliza para mover un string de caracteres de longitud variable).

- **Endoarquitectura** : Es la especificación de las capacidades funcionales de los componentes físicos, las estructuras lógicas de sus interconexiones, las interacciones, los flujos y controles de flujos de información (el programador no tiene por que saber que la instrucción MVCL requiere de un número de ciclos de lectura/grabación desde/hacia memoria no determinado a priori y que depende de la longitud del string que se desea mover).

- **Microarquitectura** : Qué componentes se abren o cierran durante la ejecución de una instrucción, en especial si tenemos una unidad de control microprogramada (en el caso MVCL se arranca un microprograma que realiza la transferencia de porciones exactas del string hasta agotar la totalidad del mismo).

Las relaciones entre estos niveles responden a la siguiente figura :

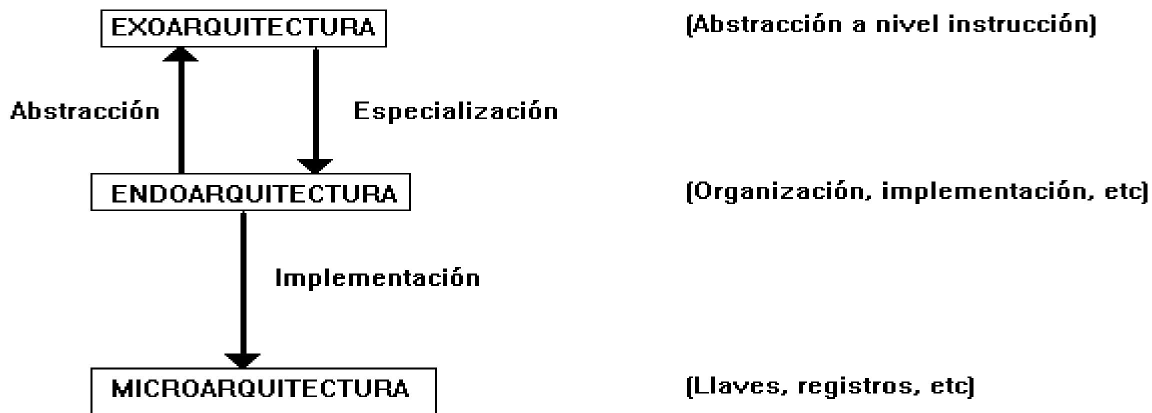


Fig. 1.1. - Niveles de Arquitecturas

1.3.3. - **Breve cuadro evolutivo**

En el siguiente cuadro mostramos muy brevemente la evolución de las diferentes Generaciones de computadoras.

Hay que destacar que el gran salto evolutivo entre la 2da y 3ra Generación está dado además por la presencia de Sistemas Operativos cada vez más potentes y entre la 3ra y la 4ta Generación por mecanismos que permitieron el mejor aprovechamiento de los recursos, como por ejemplo la memoria virtual.

La aparición de la inteligencia distribuida (workstation) y la existencia de redes de computadoras con muchas de sus funciones distribuidas a lo largo de la red están tal vez indicando la aparición de la 5ta Generación, la cual es más claramente entrevista con máquinas implementadas sobre mecanismos de inferencia.

	GENERACIONES			
	1era	2da	3ra	4ta
Procesador	Válvulas Toro magnético	Transistor Disco magnético	ICs SSI MSI	LSI VLSI memorias de semi-conductores
Estructura	Monopro-cesador	Unidades multifunción Proces. E/S	Microprocesadores MIMD (16 proc.) SIMD (64 proc vectoriales)	Workstation LAN SIMD (1024 proc.)
Velocidad	$5 * 10^{-4}$ MFLOP	$5 * 10^{-2}$ MFLOP	1 MFLOP	5 hasta 10000 MFLOP
Unidad de control	Hardwired	Hardwired	Hardware microprogramada	Hardware microprogramada
Caracteris. hardware	Aritmétic. punto fijo	Aritmética de punto flotante	Microprog. Pipeline Mem. Cache	———
Caracteris. software	Lenguaje máquina assembler No hay SO.	Leng. alto nivel. Monitores batch	Multiprogram. Mem. Virtual Sist. Operativ.	Multiprocesamiento

Fig. 1.2. - Evolución de generaciones de computadoras.

1.4. - **ARQUITECTURAS SECUENCIALES**

Casi todas las computadoras están regidas por una misma estructura general, que es la determinada por Von Neumann (1945) y presentada en su informe para su nueva computadora, la EDVAC (Computadora Electrónica de Variable Discreta).

Los conceptos de ese informe de aproximadamente 100 páginas rigen casi todos los diseños de las computadoras actuales, y son :

- Programa almacenado y Datos almacenados (esto significa que tanto programa como datos deben residir en una memoria, común o no, pero que deben estar representados internamente para poder ser ejecutados y tratados).
- Flujo de cómputo secuencial
- Representación binaria de la información (o sea, que datos y programas se representan de igual forma)
- Flujo de información interno en paralelo, en vez de en serie. (esto claramente indica que la cantidad de circuitos debe aumentar hasta tantos bits en paralelo como se pretenda que se muevan juntos en un flujo de datos).
- No paralelismo de las operaciones.

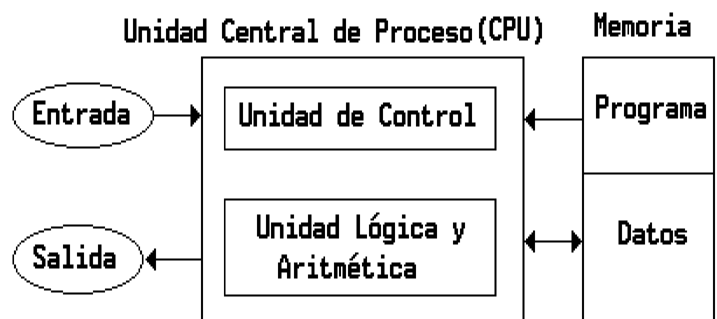


Fig. 1.3. - Arquitectura Von Neumann (1945).

Von Neumann describió cómo una computadora comprende cinco unidades (una de entrada, una de salida, un procesador aritmético, una unidad de control y una memoria) y de qué manera, siguiendo

el criterio de la unidad de control, las instrucciones se pueden almacenar en la misma memoria interna que los datos y pueden ser interpretadas por esa unidad de control de la misma forma que los datos por la unidad aritmética.

1.4.1. - **Computadoras de programa almacenado.**

En la Máquina Analítica (Babbage, siglo 19) y en sus sucesores más modernos como el Mark I de Harvard y en el ENIAC, los programas y los datos se almacenaban en memorias externas separadas.
Ingresar o alterar los programas era una tarea muy tediosa.

La idea de almacenar los programas y los datos en la misma unidad de memoria, el llamado concepto de programa almacenado, se atribuye usualmente a los diseñadores del ENIAC, y más notoriamente al matemático nacido en Hungría John Von Neumann (1903-1957) quien fue un consultor en el proyecto ENIAC.

Además de facilitar la programación, el concepto de programa almacenado hizo posible que un programa modificara sus propias instrucciones.

1.4.2. - **Proceso de Cómputo Secuencial**.

El concepto más difundido cuando se habla de una arquitectura Von Neumann es el del proceso de cómputo.

En estas arquitecturas el programa se almacena en memoria como una secuencia de instrucciones y se ejecuta extrayendo las mismas en secuencia e interpretándolas. Por lo tanto el curso que sigue el cómputo viene dado por la secuencia de las instrucciones del programa (es decir, el flujo de control del programa).

Nótese que uno de los elementos característicos de toda arquitectura Von Neumann es el llamado Registro de Próxima Instrucción (del inglés Program Counter, Contador de Programa) cuya misión consiste en indicar en todo momento cuál es la siguiente instrucción que deberá ejecutarse respetando la secuencia.

1.4.3. - **Arquitectura Básica de un monoprocesador**

Un típico monoprocesador cuenta con tres componentes básicos: la memoria principal, la unidad central de proceso (CPU, que generalmente cuenta con por lo menos una unidad aritmético-lógica -UAL- y una unidad de control), y el subsistema de E/S (con unidades para entrada y para salida o para entrada/salida).

Según John Backus (Agosto 1978), en su forma más simple una computadora Von Neumann consta de tres partes: una unidad central de proceso (CPU), una unidad de almacenamiento y un tubo de conexión que puede transmitir una sola palabra entre la CPU y el almacenamiento (y enviar una dirección desde almacenamiento).

Denominemos a este tubo el *Cuello de botella de Von Neumann.*

El trabajo de un programa consiste en alterar el contenido del almacenamiento de alguna forma; cuando uno piensa que dicha tarea debe realizarla íntegramente bombeando a través del cuello de botella de Von Neumann se da cuenta inmediatamente de la razón de tal nombre.
Irónicamente, una gran parte del tráfico en el cuello de botella no son datos útiles, sino más bien nombres de datos (direcciones), así como operaciones y otros datos que sirven solamente para obtener o calcular tales nombres (instrucciones).

Antes de que un dato sea enviado por el tubo hacia la CPU su dirección debe estar en la CPU; esa dirección llegó allí porque fue enviada por el tubo desde el almacenamiento hacia la CPU o bien, fue generada por la CPU a través de alguna operación.

Si la dirección proviene del almacenamiento entonces su dirección debió provenir del almacenamiento o fue generada por la CPU, y así siguiendo.

Si, por otra parte, la dirección fue generada en la CPU, debió ser generada por alguna regla FIJA (por ejemplo: sumar 1 al contador de programa) o por una instrucción que provino del almacenamiento a través del tubo, en cuyo caso su dirección debió provenir .... y así siguiendo.

1.4.4. - **Unidades funcionales**

El objetivo de todo esto, o sea el objetivo de una computadora, es que un algoritmo, escrito en una determinada codificación, que constituye el programa y cuyas instrucciones están almacenadas en memoria, instruyan a la Unidad de Control, tomen datos y los transformen en resultados.

De aquí resulta bien claro cuáles son las funciones de las Unidades Funcionales, o sea :

- En la memoria se almacenan datos y programas, sus instrucciones informan a la Unidad de Control qué hacer, por ejemplo, que tome un dato de la Unidad de Entrada lo almacene en memoria, lo lleve al procesador, lo transforme por medio de la Unidad Aritmético/Lógica, lo almacene nuevamente en memoria ya transformado y lo devuelva al medio externo por una Unidad de Salida.
- Las instrucciones son tomadas una a una y en secuencia, obviamente existen instrucciones para alterar esa secuencia, pero es exclusivamente para comenzar una nueva.

En la Fig. 1.4 podemos ver la estructura general de la típica computadora de la serie IBM S/360.

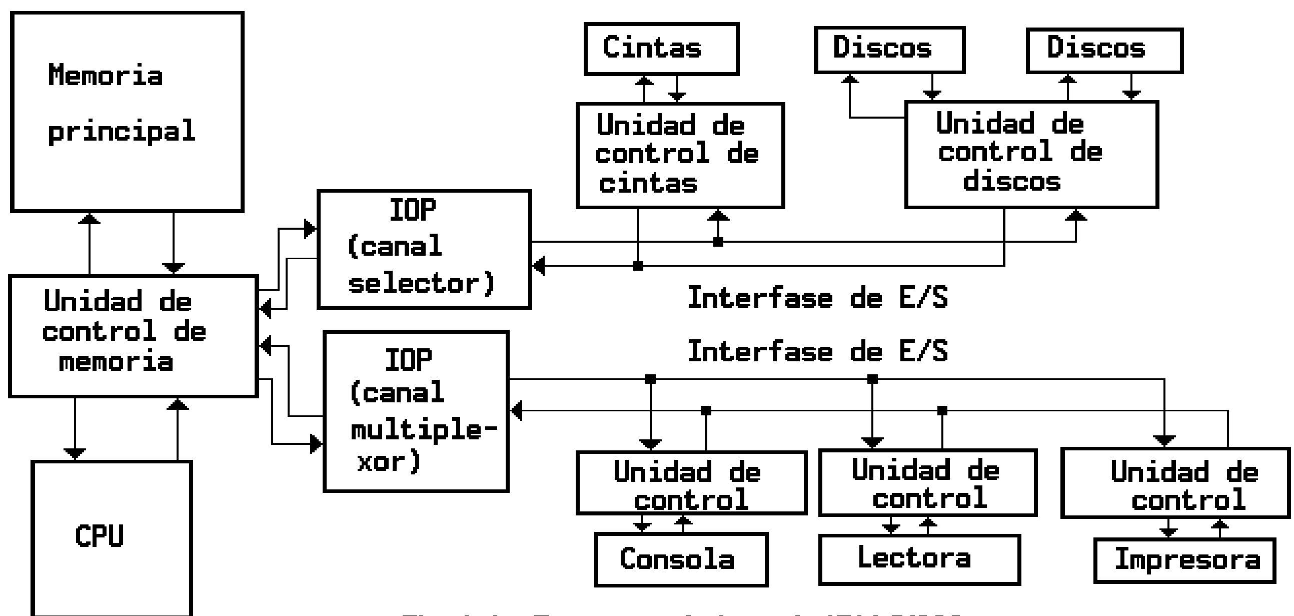


Fig. 1.4. - Estructura de la serie IBM S/360.

La misma utiliza dos tipos de Procesadores de E/S : canales multiplexores y canales selectores.

Los canales multiplexores pueden intercalar (multiplexar) la transmisión de datos entre memoria principal y diferentes dispositivos de E/S, en tanto que sólo un dispositivo de E/S conectado a un canal selector puede transmitir o recibir información en un momento dado (Ver Capítulo 3).

Los canales selectores se utilizaron para dispositivos de muy alta velocidad, por ejemplo : cintas o discos magnéticos, en tanto que los multiplexores sirven para dispositivos de baja velocidad (impresoras, lectoras de tarjetas, etc).

Actualmente se utilizan los canales block multiplexor para dispositivos de muy alta velocidad.

Cada procesador de E/S se encuentra conectado a un bus que se denomina la **interfase de E/S** compuesta por un conjunto de líneas de datos y líneas de control. Este bus de interfase es compartido por todos los periféricos que pertenecen a un determinado procesador de E/S.

Un dispositivo de E/S o un conjunto de dispositivos de E/S del mismo tipo está supervisado localmente por una unidad de control que es particular para el tipo de dispositivo en cuestión.

En las Fig. 1.5 y 1.6 se pueden observar dos arquitecturas típicas de monoprocesadores.

En la figura del monoprocesador VAX puede verse que:

La CPU contiene :

- 16 registros de uso general, uno de ellos (register status) registra el estado actual del procesador y del programa que está siendo ejecutado; además existe un registro adicional que sirve como la PC (palabra de control) del programa,.
- Una unidad Aritmética y Lógica con punto flotante.
- Una memoria cache local con una memoria de diagnóstico que es optativa.

La CPU, la memoria principal y el subsistema de E/S están todos conectados a un bus sincrónico de interconexión, el synchronous backplane interconnect (SBI).

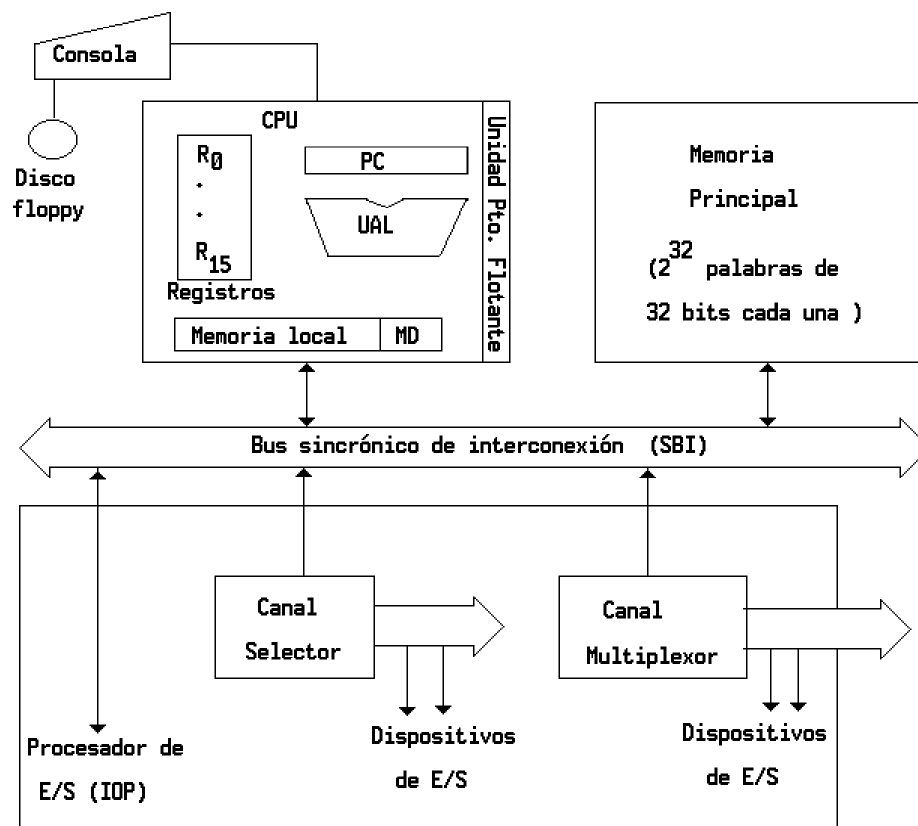


Fig. 1.5. - Arquitectura de sistema de la supermini Vax-11/780 monoprocesador.

A través de este bus todos los dispositivos de E/S se pueden comunicar entre sí, con la CPU o con la memoria.

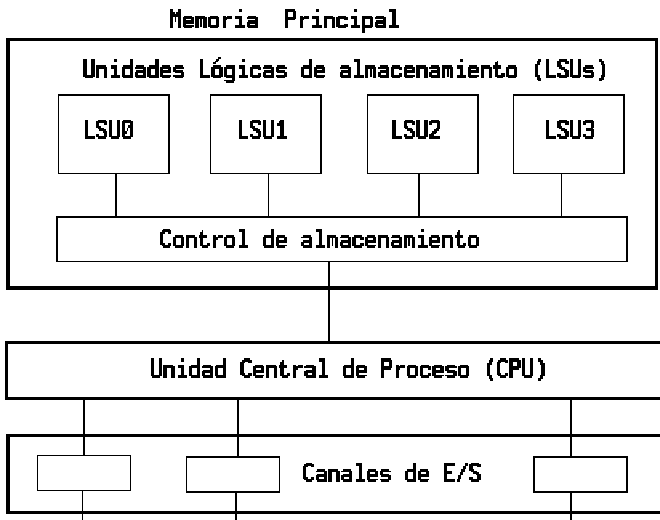


Fig. 1.6. - Arquitectura del mainframe IBM S/370.

Los periféricos pueden estar conectados a este bus por medio de canales selectores o multiplexores o a través de Procesadores específicos de E/S.

En la arquitectura del sistema 370 :

- La CPU contiene el decodificador de instrucciones (normalmente dentro de la Unidad de Control), la unidad de ejecución y también una memoria cache.
- La memoria principal está dividida en cuatro unidades llamadas Unidades Lógicas de Almacenamiento (Logical Storage Unit LSU), que pueden accederse en forma alternada. El controlador de almacenamiento provee una conexión de tipo multiple-port (multipuerta) entre la CPU y las 4 LSU.
- Los periféricos están conectados al sistema vía canales de alta velocidad que operan en forma asincrónica con la CPU.

En la Fig. 1.7 podemos apreciar, por un lado cómo mejoran las velocidades (tomando como parámetro que la velocidad del modelo 30 es 1 el modelo 70 lo aventaja 50 veces) respecto de dos modelos de la serie S/360 antes mencionada. Pero, por otra parte, es interesante observar como existe una descompensación entre la velocidad de la memoria y la velocidad de la CPU. Es esta diferencia de velocidades lo que nos permitirá más adelante el analizar los mecanismos mediante los cuales se busca equilibrar esta diferencia de velocidades de los componentes de un computador (Bandwidth).

Subsistema	Parámetros	Modelo 30	Modelo 70
Memoria Principal	Ciclo de Memoria	2μs	1μs
	Ancho del bus	8 bits	64 bits
	Velocidad máxima de transferencia	4 * 10^6 bits/s	128 * 10^6 bits/s
CPU	Ciclo de CPU	30 ns	6 ns
	Tecnología de los registros de trabajo	Toro	Semiconductor
	Ancho del bus de CPU	8 bits	64 bits
	Velocidad de cómputo relativa	1	50

Fig. 1.7. - Comparativa de los parámetros de diseño de dos modelos de la serie IBM S/360.

En todas las arquitecturas se hace necesario lograr un balance entre los distintos subsistemas a efectos de evitar los cuellos de botella y mejorar el THROUGHPUT total del sistema (que es la cantidad de trabajos realizados

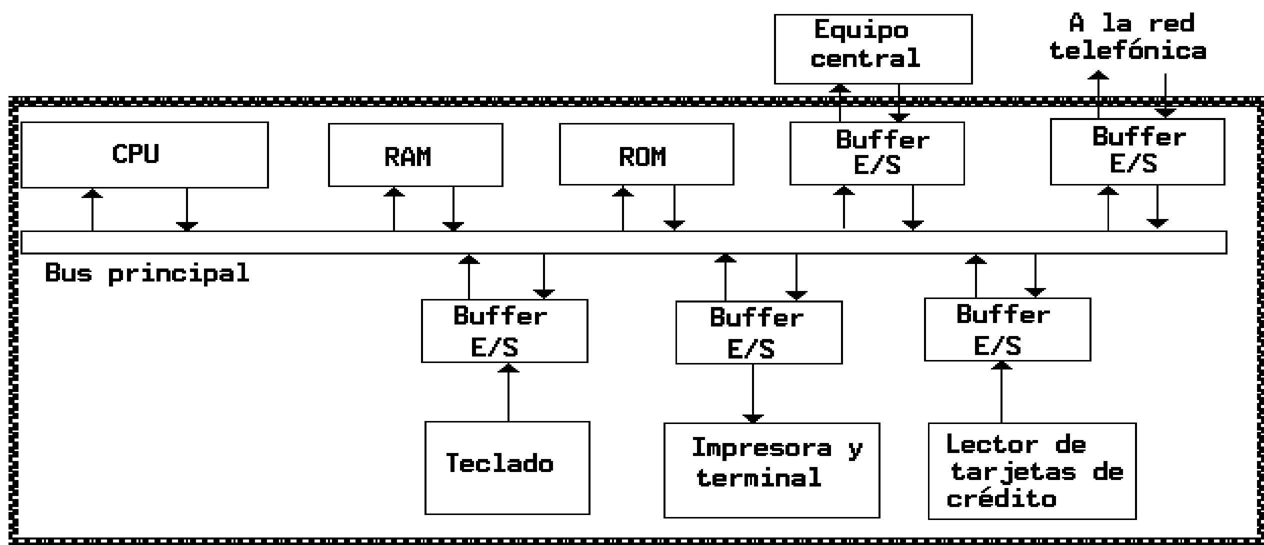


Fig. 1.8. - Computador de propósito específico Punto de Ventas (Point-of-Sales).

por unidad de tiempo).

Existe un número bastante grande de arquitecturas diferentes de acuerdo a la aplicación a la cual se destina el computador en sí. Por ejemplo en la Fig. 1.8 se aprecia la arquitectura de un computador de propósito específico. En este caso en particular se trata un microcomputador de tipo POS (Point-Of-Sales o Punto de ventas) que se han popularizado mucho en los últimos años a raíz de las redes de teleprocesamiento que permiten que un comerciante tenga un instrumento que le permite verificar y actualizar sus movimientos de ventas a clientes.

1.5. - **IMPLEMENTACIÓN REAL DE LAS UNIDADES FUNCIONALES DE UN COMPUTADOR**

1.5.1. - **PROCESADOR O CPU**

Se define la CPU o procesador como la unidad funcional que ejecuta las instrucciones de una determinada arquitectura de propósito general. Muchos sistemas de computación tienen un procesador que se encarga de la totalidad de las funciones de interpretación de instrucciones y su ejecución.

El término de propósito general sirve para diferenciar a la CPU de otros procesadores tales como los procesadores de E/S cuyas funciones son en cierta forma más restringidas o para un propósito específico.

Generalmente existe una sola CPU en la mayoría de las computadoras. Una computadora con una sola CPU se denomina Monoprocesador; una computadora con más CPUs se denomina un Multiprocesador.

Cualquier CPU muestra una bien clara división entre lo que es el procesamiento de los datos y el control del procesamiento en sí. La primer función suele asignársele a una Unidad Aritmético y Lógica (UAL o ALU en inglés), en tanto que la segunda pertenece a la Unidad de Control de Programa o Unidad de Instrucciones.

La Fig. 1.9 muestra un bloque sencillo que se utiliza para representar a la CPU. Existe un bus de datos de una sola palabra que es el camino por el cual fluye la información desde o hacia la CPU.

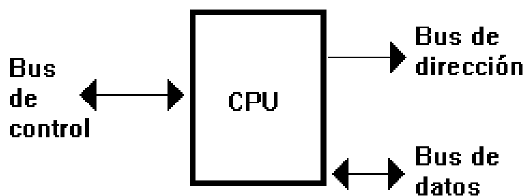


Fig. 1.9. - Una Unidad Central de Proceso.

Generalmente existe además un segundo bus por el cual se transmiten las direcciones desde la CPU a la memoria principal y también a los dispositivos de E/S.

Existen además algunas líneas de Control que se utilizan para controlar los otros componentes del sistema y además para sincronizar sus operaciones con las de la CPU.

La Fig. 1.10 muestra la estructura de la CPU del sistema S/360-370. La unidad aritmético y lógica está dividida en tres subunidades que cumplen las siguientes funciones :

- Operaciones de punto fijo, que incluye aritmética entera y cálculo efectivo de las direcciones.
- Operaciones de punto flotante.
- Operaciones de longitud variable, que incluye aritmética decimal y operaciones sobre strings de caracteres.

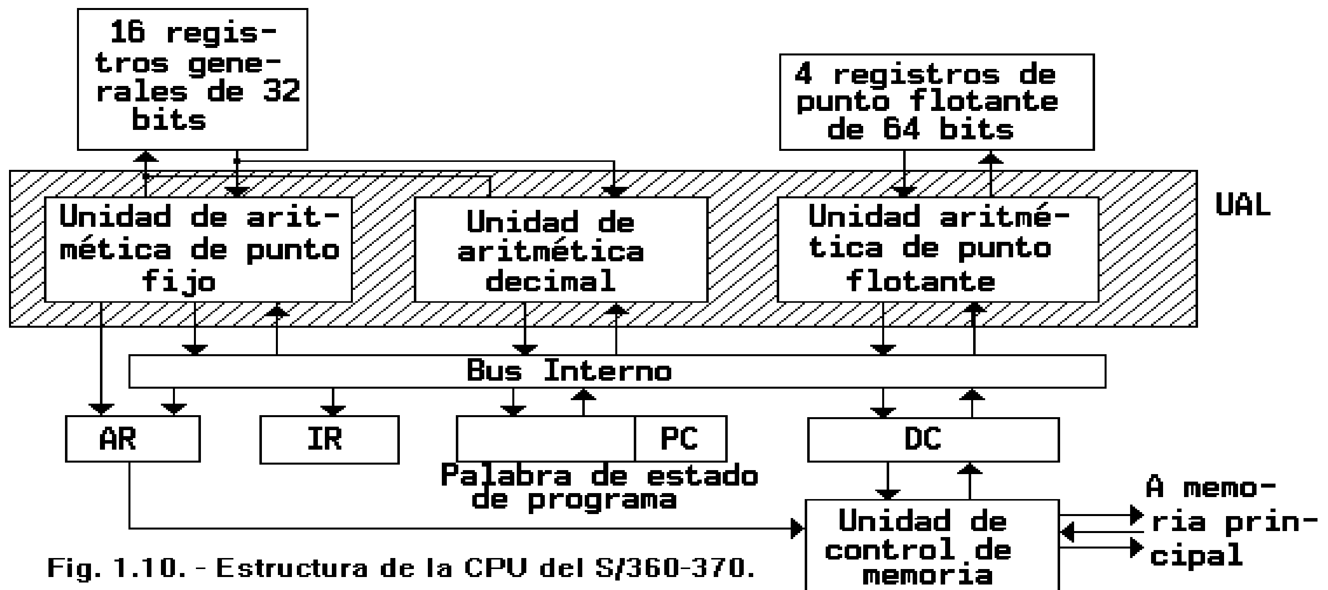


Fig. 1.10. - Estructura de la CPU del S/360-370.

Existen dos conjuntos (cadenas) independientes de registros para almacenar los datos y las direcciones. Los 16 registros generales se usan para almacenar los operandos y los resultados y además pueden usarse de índices. Los 4 registros de punto flotante se usan para la aritmética de punto flotante.

Los registros DR (registro de dato), AR (registro de dirección) e IR (registro de instrucción) son standard.

La Palabra de estado de programa (Program Status Word PSW) almacenada en un registro especial indica el estado del programa, las interrupciones a la cuales la CPU puede responder, y la dirección de la próxima instrucción que se deberá ejecutar (almacenada en el PC -program counter- o registro de próxima instrucción). La razón primordial de la existencia de la PSW es por el manejo de las interrupciones.

La CPU se encuentra en un momento dado en alguno de varios estados. Cuando ejecuta una rutina del sistema operativo, es decir que es el sistema operativo el que tiene el control explícito sobre la CPU, se dice que se encuentra en estado **supervisor** (o modo maestro). Ciertas instrucciones solo pueden ejecutarse en esta modalidad. Generalmente la CPU se encuentra en estado **problema** (o modo esclavo) mientras ejecuta programas de los usuarios. El estado de la CPU se refleja en la PSW.

La PSW contiene además una clave que se utiliza como **protección de memoria**. La memoria principal está dividida en bloques (de por ejemplo 2K), cada uno de los cuales tiene asignado un número de clave. La clave de almacenamiento especifica el tipo de operación que está permitido realizar sobre ese bloque (lectura, lectura y grabación, ni lectura ni grabación). Una operación que requiera un acceso a un determinado bloque de memoria se ejecuta solamente si la clave de protección del bloque que se desea acceder coincide con la almacenada en la PSW de ese instante.

1.5.2. - **MEMORIA**

La memoria de un sistema de computación se divide en dos mayores subsistemas :

- La Memoria Principal, que consiste en dispositivos relativamente veloces conectados directamente y controlados por la CPU.
- La Memoria Secundaria, con dispositivos de menor velocidad y de menor costo que se comunican indirectamente con la CPU (o no) a través de la memoria principal.

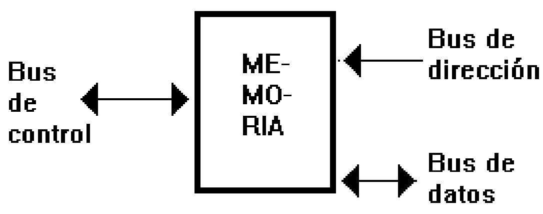


Fig. 1.11. - Una Unidad de Memoria.

Los dispositivos de memoria secundaria tales como discos magnéticos o cintas magnéticas se utilizan para almacenar grandes cantidades de información. Muy a menudo se encuentran bajo el control directo de procesadores de propósito específico (IOPs input-output processor, procesadores de E/S). Esta memoria secundaria se considera generalmente como parte del subsistema de E/S.

En muchos casos la memoria está organizada en base a palabras direccionables. Esto significa que la información puede ser accedida de a una palabra por vez. La ubicación de cada palabra tiene asociada una dirección que la identifica unívocamente.

Para acceder a una palabra en particular la CPU envía su dirección y una apropiada función de control (Leer o Grabar) a la memoria. Si la función es de grabación además la CPU debe colocar la palabra que desea grabar en el bus de datos de la memoria desde donde será transferida a su correcta ubicación.

1.5.2.1. - **Organización de la Memoria**

Por ahora supondremos que la memoria está compuesta por celdas direccionables y que por cada dirección se acceden n bits (palabras) (8 ó ... 64 ó ...etc) donde no es posible diferenciar instrucciones de datos.

El procesador las diferenciará y las tratará como instrucciones si la dirección proviene de la PC, y como datos si la dirección proviene del operando de la instrucción.

1.5.2.2. - **Formato de Instrucciones**

Los formatos de las instrucciones pueden ser varios, veremos rápidamente algunos para el ejemplo C = A + B.

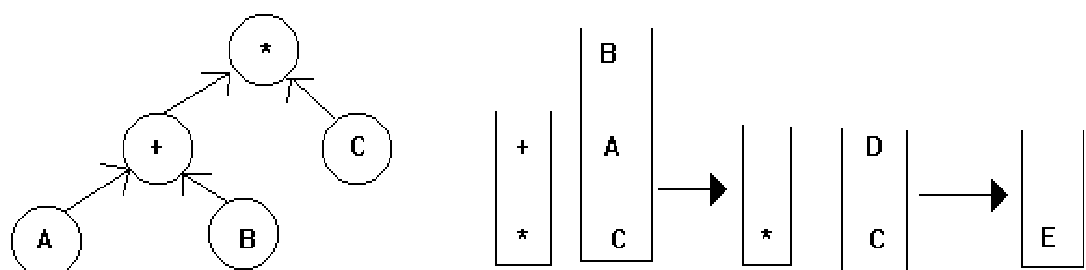


Fig. 1.12.

4 op. ADD A, B, C, D (con D dirección de la próxima instrucción)
3 op. ADD A, B, C
2 op. MOVE C, B Mueve B a C / ADD C, A El resultado queda en C
1 op. LOAD A Carga en acum. o reg. / ADD B Suma Acumulador + B
STORE C Pone el resultado del acumulador en C
0 op. Trabaja con Stacks (Pilas) (sus direcciones están implícitas)
Ejemplo : (A + B) * C

El operador se retira de un stack y los dos operandos del tope del otro stack, y su resultado va al tope del stack de operandos.

1.5.2.3. - **Modos de direccionamiento a Memoria.**

Existen varios tipos de modos de direccionamiento a memoria, aquí veremos rápidamente algunos de ellos.

- Modo Directo : la dirección de los operandos contienen los datos.
- Modo inmediato : el dato está en la instrucción.
- Modo Indirecto : dirección de dirección
- Modo Indexado : dirección + índice, da la dirección

Pueden existir variantes y combinaciones de estos modos de direccionamiento. Además son dependientes de la organización de la memoria, según veremos en el capítulo de Memoria.

1.5.3. - **DISPOSITIVOS DE E/S**

Los dispositivos de E/S son el medio por el cual un computador se comunica con el mundo exterior. La función primaria de la mayoría de los dispositivos de E/S es actuar como traductores, por ejemplo, convirtiendo información de una forma de representación física en otra.

1.5.3.1. - **Interconexión de Unidades Funcionales. Estructura de Bus**

Las unidades funcionales deben estar interconectadas en forma organizada, por medio de cables (vías) que transportan señales, (bits de datos, direcciones, etc) a estos conjuntos de cables se los denomina **buses** (se pueden encontrar en la bibliografía también con los nombres : red de conmutación (switching network), controlador de comunicaciones y controlador de buses).

La función primaria de los buses es establecer una comunicación dinámica entre los dispositivos que se hallan bajo su control.

Por razones de costo los buses se comparten. Esto lleva a producir lo que se denomina **contención** en el bus, cuando más de un componente desea hacer uso del él al mismo tiempo. El bus debe resolver tal contención seleccionando uno de los pedidos en base a algún tipo de algoritmo (por ejemplo, prioridad) y encolando el resto a la espera de su liberación.

Los requerimientos simultáneos a un mismo dispositivo surgen del hecho de que la comunicación entre los diferentes componentes del sistema es generalmente de tipo asincrónica.

Esto se puede atribuir a diversas causas :

- Existe un alto grado de independencia entre los componentes. Por ejemplo, la CPU y los IOPs ejecutan diferentes programas e interactúan frecuentemente en momentos impredecibles.
- Las velocidades de los diferentes componentes varían dentro de un amplio rango. Por ejemplo la CPU opera de a 1 a 10 veces más velozmente que los dispositivos de memoria, y esta a su vez es varias veces más veloz que los dispositivos de E/S.
- La distancia física que separa los componentes puede ser grande de manera tal que sea imposible lograr una transmisión sincrónica de la información entre ambos.

La transmisión asincrónica se implementa muy frecuentemente con un mecanismo que se denomina **"handshaking"** (acuerdo). Supongamos que tenemos que trasmitir un dato desde el dispositivo A hacia el B. A coloca el dato en cuestión en el bus de datos de A a B y envía luego una señal de control hacia B, que se denomina generalmente **"ready"**, para indicarle la presencia del dato en el bus. Cuando B reconoce la señal de ready, transfiere el dato desde el bus hacia un área propia y luego activa una línea de control de **"acknowledge"** (toma conoci
miento) hacia A. Cuando A recibe la señal de acknowledge comienza la transmisión del siguiente dato. Luego una secuencia de ready/acknowledge acompaña la transferencia de los datos haciéndola independiente de las velocidades de los dos componentes.

Las líneas de datos y de control conectadas a un dispositivo así como la secuencia de señales requeridas para comunicarse constituyen la **"interfase"** del dispositivo. Cuanto más similares sean las interfases de los dispositivos mucho más sencillo será que se puedan comunicar. Si se utilizan interfases standard en todo el sistema, se incrementa muchísimo la facilidad con la cual el mismo puede expandirse o modificarse.

Veremos a continuación las tres estructuras más usadas de buses.

1.5.3.2. - **Dos Buses**

Todo dato de E/S pasará por el procesador, luego las operaciones y transferencias de E/S se realizan bajo control directo del procesador, o sea la E/S es controlada por programa.

Fig. 1.13.

1.5.3.3. - **Dos Buses (Alternativa) (Canales)**

Aquí las operaciones de E/S y transferencia se realizan bajo control de procesadores de E/S (canales de E/S).

Fig. 1.14.

Generalmente el procesador pasa a los canales los parámetros y el control de las operaciones de E/S.

Aquí surgen los problemas de acceso a memoria simultáneos, los que deben ser resueltos, una forma es por medio de una sola entrada (Fig. 1.15).

En este caso se produce el llamado robo de ciclos, o sea el canal roba ciclos de acceso a memoria al procesador.

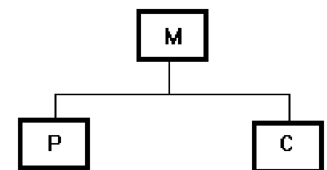


Fig. 1.15.

Puede existir algún control de acceso a memoria tipo MMU (Memory Management Unit) que evite la posibilidad de accesos simultáneos a una misma posición de memoria (Fig. 1.16).

Otra forma de controlar este problema es por medio de la técnica de Interleaving (ver Capítulo 2), también con un controlador de memoria, pero que permite la entrada simultánea a memoria sobre distintas porciones (franjas o bancos).

Aumentando la cantidad de buses obviamente se aumenta la capacidad de transferencia simultánea de información, este caso como los dos anteriores es lo que generalmente se conoce como estructura **Multibus**.

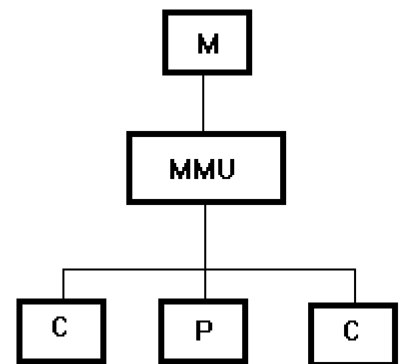


Fig. 1.16.

1.5.3.4. **Un bus**

Aquí el bus se puede utilizar para una sola transferencia por vez (obvio, hay un solo bus) o sea que solo dos unidades funcionales pueden estar haciendo uso de él.

Es fácil incorporar nuevos periféricos.

Es una estructura de bajo costo y es muy utilizada en microcomputadoras.

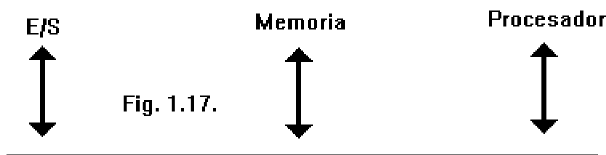


Fig. 1.17.

A pesar de estas diferencias de estructura que hacen al rendimiento, los principios fundamentales de funcionamiento son independientes de ellas y siguen siendo máquinas secuenciales.

1.5.4. - **ORGANIZACION DE PROCESADORES**

1.5.4.1. - **Procesador de un Bus**

En el caso de un procesador de un solo bus la ALU y todos los registros del procesador están conectados a través de un único bus.

En la Fig. 1.18 podemos ver este esquema donde :

RDM : Registro de dirección de Memoria
RBM : Registro buffer de memoria
R_0 a R_n = son registros de uso general
PC = Program Counter
J = es un registro
Z = es un registro acumulador

RI = Registro de instrucción

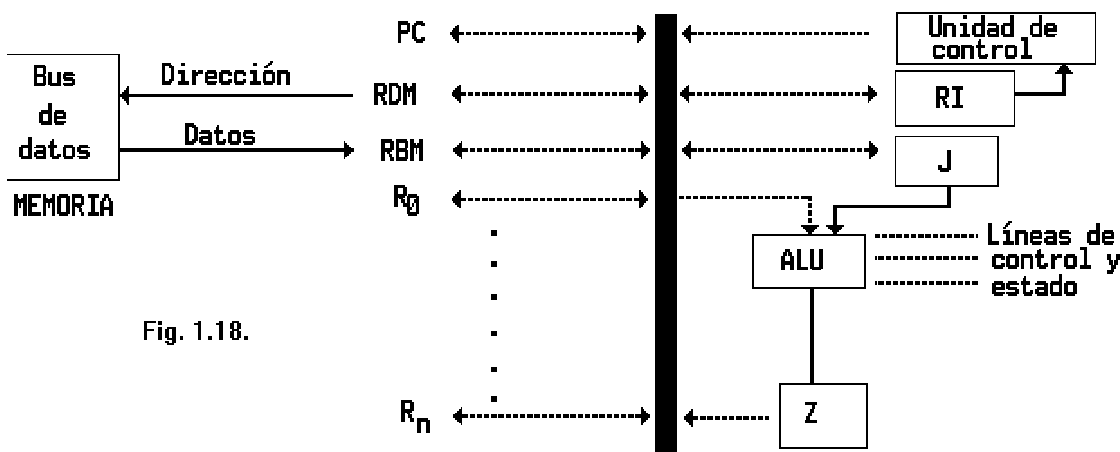


Fig. 1.18.

Para ejecutar la instrucción ADD R1,Op (sumar Op + R1 dejando el resultado en R1) (suponemos instrucciones de igual longitud, y mientras se pone la dirección de lectura, ya se incrementa el RPI) primero se levanta la instrucción propiamente dicha haciendo:

- PC out, RDM in, Read (memoria), Clear J, Carry = 1, ADD, Z in .

Luego la ejecución propiamente dicha será :

- Z out, PC in, Espera (Función de Memoria)
- RBM out, RI in
- Campo Dir RI out, RDM in, Read (memoria)
- R1 out, J in, Espera (función de memoria)
- RBM out, ADD, Z in
- Z out, R1 in

Si lo hacemos para la instrucción ADD R2, R1 (sumar R1 + R2 dejando el resultado en R2) la carga de la instrucción será :

- PC out, RDM in, Read (Memoria), Clear J, Carry = 1, ADD, Z in

y luego, la ejecución :

- Z out, PC in, Espera (Función de Memoria)
- RBM out, RI in
- R2 out, J in
- R1 out, ADD, Z in
- Z out, R2 in

1.5.4.2. - **Procesadores Multibus**

Veamos sobre el ejemplo de la Fig. 1.19 la operación ADD R2, R1.

La carga de instrucción será :

- PC out, G (hab), RDM in, Read (Memoria), J in

y la ejecución entonces :

- Carry = 1, ADD, PC in, Espera (Función de Memoria)
- RBM out, G (hab), RI in
- R1 out, G (hab), J in
- R2 out, ADD, ALU out, R2 in

Con lo cual vemos que esta suma se redujo a un "tiempo" con respecto al anterior, o sea, si tenemos más caminos (buses) más operaciones en paralelo se pueden realizar.

Si por ejemplo se tiene una estructura como la de la PDP-11 (Fig. 1.20).

La ejecución de la operación ADD R3, R2, R1 resultaría en la ejecución de sólo:

- R1 out_A,

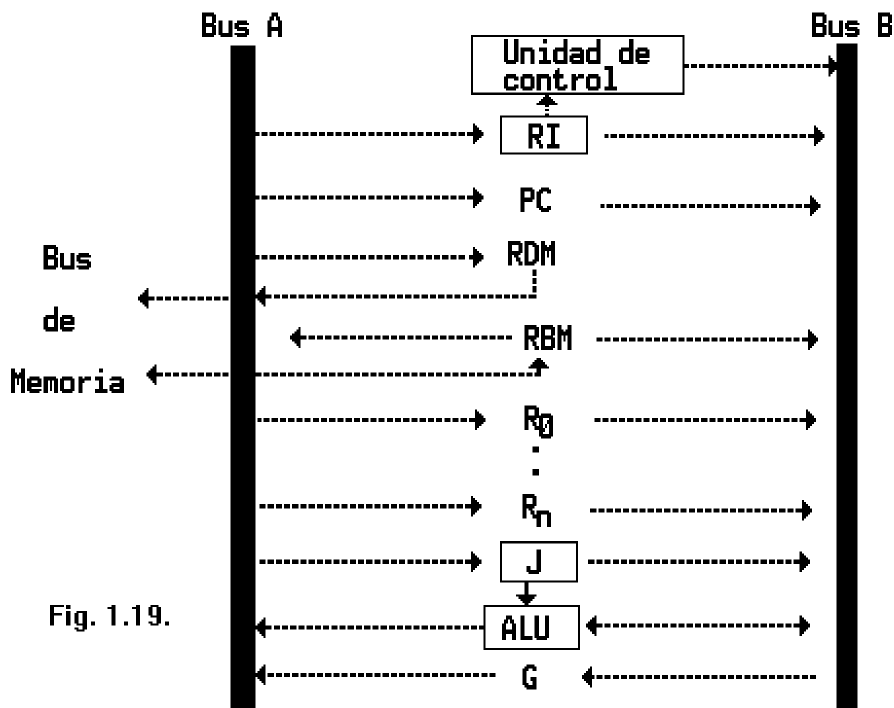


Fig. 1.19.

- A in_A,
- R2 out_B,
- B in_B,
- A out,
- B out,
- ADD,
- ALU out_C,
- R3 in_C

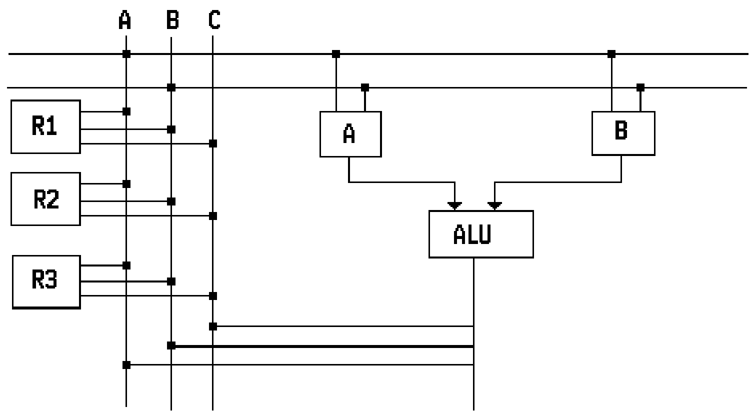


Fig. 1.20. - Estructura de la PDP-11.

1.5.5. - **UNIDAD DE CONTROL (Procesador)**

Si observamos la ejecución de las instrucciones anteriores, veremos que ésta consta de una serie de pasos, en los cuales es necesario dar tiempo para que se completen las operaciones de cada una de ellas.

Estos tiempos (que por razones de simplicidad los tomamos como la unidad y cada paso tarda una unidad) deben ser generados por un contador activado por un reloj.

La unidad de control deberá entonces **no** sólo decodificar un código de operación, sino también emitir señales (códigos de condición) según se vayan ejecutando las instrucciones (overflow, carry -acarreo-, cero, negativo, etc). Es decir la Unidad de Control recibe de la Unidad Aritmético-Lógica señales que indican el estado que ha resultado de la ejecución de la operación indicada.

Nótese por ejemplo que existen ciertas instrucciones que permiten realizar una cierta operación independientemente de si se produce o no overflow; en tales casos al recibir la señal de overflow de la ALU la Unidad de Control no toma ninguna acción, ni de recupero del error (si fuera esto posible) ni indicando que se ha producido una falla debido a que el código de operación de la acción solicitada específicamente indica que tal condición no debe atenderse.

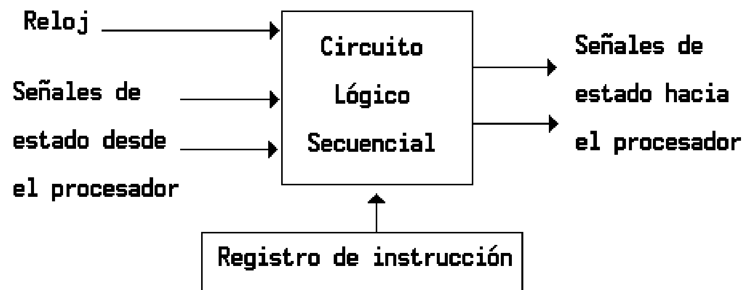


Fig. 1.21. - Circuito de control hardwired.

1.5.5.1. - **Microprogramación**

La microprogramación es una técnica para implementar la función de control de un procesador de una manera flexible y sistemática.

El concepto fue enunciado por primera vez por Maurice V. Wilkes en 1951, y fue implementado por primera vez en algunos de los computadores de primera y segunda generación.

Sin embargo no fue hasta mediados de los años 60 en los que su aparición en algunos modelos de las series IBM S/360 determinó su mayor difusión.

La microprogramación debe ser considerada como una alternativa del control de circuitos hardwired (o cableado).

Un circuito de control hardwired es típicamente un circuito secuencial como puede verse en la Fig. 1.21. Un circuito de control microprogramado tiene la estructura que puede apreciarse en la Fig. 1.22.

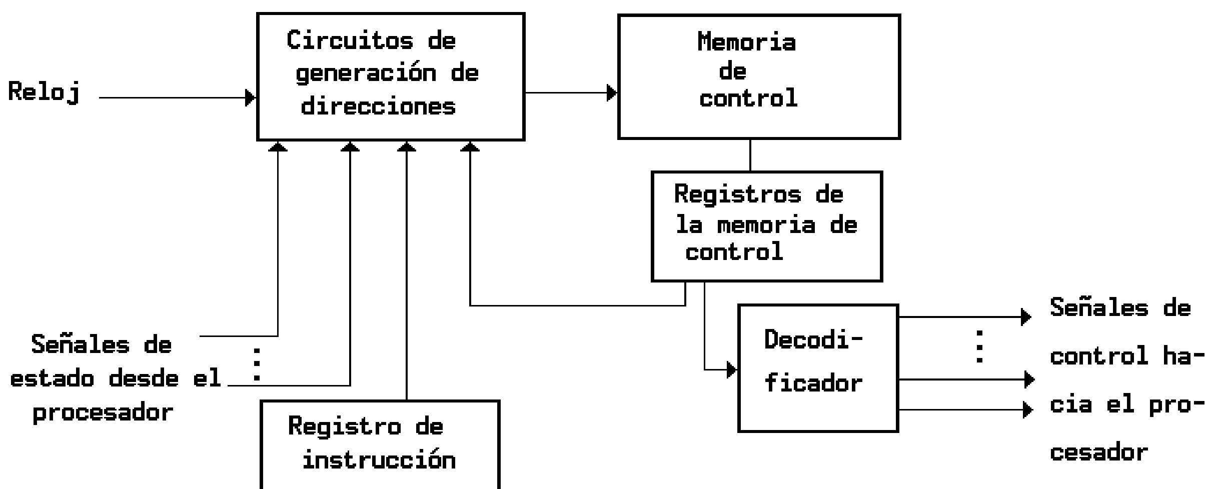


Fig. 1.22. - Circuito de control microprogramado.

En el caso de una unidad de control implementada en hardware será un conjunto de circuitos que interpretan códigos de operación y generan señales dependiendo de los estados de cada instrucción.

En la microprogramación cada instrucción que se encuentra bajo el control del procesador desencadena una secuencia de microinstrucciones llamada microprograma, que se obtienen de una memoria especial de acceso random (al azar) denominada memoria de microprogramas.

Las microinstrucciones indican la secuencia de microoperaciones o las transferencias entre registros necesarias para interpretar y ejecutar la instrucción madre.

Cada instrucción obtenida de la memoria principal del sistema inicia una secuencia de microinstrucciones obtenidas de la memoria de microprogramas.

Para ilustrar su funcionamiento tomemos como ejemplo los 3 últimos pasos de la instrucción ADD R2, R1 (última suma del punto 1.5.4.1))

R1 in	R1 out	R2 in	R2 out	J in	J out	Z in	Z out	ADD...
0	1	0	0	1	0	0	0	0
0	0	0	1	0	0	1	0	1
0	0	1	0	0	0	0	1	0

Donde lo anterior representa los pasos de control para poder ejecutar esa porción de instrucción.

Cada una de las líneas recibe el nombre de microinstrucción y un conjunto de microinstrucciones que corresponden a la resolución de una instrucción, recibe el nombre de microprograma de la instrucción.

La microprogramación provee una forma sistemática para el diseño de circuitos de control, ya que permite estandarizar el control incluyéndolo directamente en el hardware de la máquina deseada.

Potencia grandemente la flexibilidad de una computadora. El conjunto de instrucciones de una máquina microprogramada puede cambiarse simplemente cambiando la memoria de control.

Esto hace posible que una computadora microprogramada ejecute directamente programas escritos en otros lenguajes de máquina o en una computadora diferente, proceso denominado emulación.

Las unidades de control microprogramado tienden a ser más costosas y más lentas que las de tipo hardwired porque la ejecución de una instrucción implica varios accesos a la memoria de microprogramas, o sea este tiempo de acceso es importante en el tiempo de ejecución de una instrucción.

Debido a la estrecha interacción entre el software y el hardware en las unidades microprogramadas se suele referir a los microprogramas como firmware.

EJERCICIOS

1) En base a qué se define o describe una arquitectura ?
2) Cuáles son las unidades funcionales de un computador ?
3) Cuáles son las características distintivas de las 4 grandes generaciones de computadores ?
4) Puede existir concurrencia en un sistema con una única CPU ?
5) Cuál es el nivel de arquitectura que especifica las capacidades funcionales de los componentes físicos de un computador ?
6) Cuál es la función de la UC ?
7) La palabra de estado de programa (PSW) según sus funciones pertenece al componente hardware UC. Justifique el porqué.
8) Cómo funciona un mecanismo de comunicación handshaking ? Es sincrónico o asincrónico ?
9) Cuál es la diferencia entre una UC microprogramada y otra hardwired ?
10) Cómo es una estructura de interconexión de dos buses ?
11) Cuál de las siguientes implementaciones es correcta y porqué :
- Un único bus conecta la memoria, la CPU y los canales
- El acceso a memoria desde los periféricos es siempre a través de la CPU.

12) Cuál es el inconveniente típico de un procesador de un bus común ?
13) Cuáles son los dos conceptos que se manejan cuando uno se refiere a una máquina de arquitectura Von Neumann ?
14) Qué es un microprograma y para qué se utiliza ?
15) Qué es Throughput ?

CAPITULO 2

MEMORIA

2.1. - SUBSISTEMA DE MEMORIA

Este subsistema está constituido por:
i) Dispositivos de almacenamiento capaces de contener instrucciones y datos requeridos por ellas.
ii) Los algoritmos necesarios para el control y manejo de la información almacenada. Estos algoritmos pueden estar implementados por hardware y/o software.

Los distintos dispositivos o componentes de este subsistema pueden ser jerarquizados según el cuadro 2.1, donde se muestra la variación de velocidad/capacidad entre las distintas jerarquías. Las memorias pueden clasificarse o jerarquizarse en base a diferentes atributos.

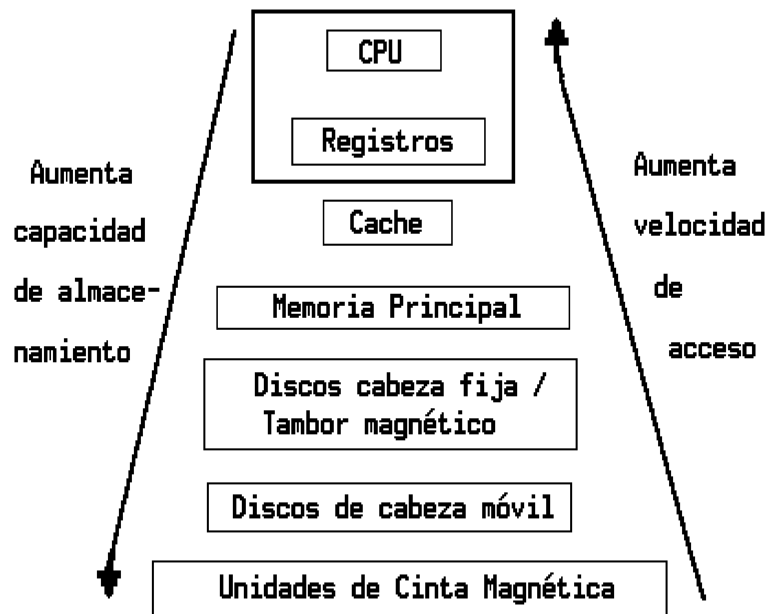


Fig. 2.1. - Jerarquías de memorias.

2.1.1. - Clasificación en base al método de acceso

Esta clasificación tiene en cuenta el orden o la secuencia en la cual la información puede ser accedida. Una memoria es aleatoria si una determinada posición de la misma puede accederse independientemente de la posición accedida con anterioridad; este es el caso de las memorias de semiconductores o de las memorias ferro-magnéticas (de núcleos).

Las memorias de acceso serial o secuencial son aquellas en donde una determinada posición puede ser accedida solamente en cierta secuencia predeterminada; este es el caso de cintas magnéticas o de burbujas magnéticas.

Las memorias semialeatorias o de acceso directo están constituidas por una serie de pistas que pueden accederse aleatoriamente, pero la posición deseada dentro de dichas pistas se accede en forma serial; este es el caso de discos magnéticos (con cabeza fija o móvil) y tambores magnéticos.

2.1.2. - Clasificación en base a velocidad o tiempo de acceso

La memoria interna del procesador comprende un conjunto reducido de registros de alta velocidad usados como registros de trabajo del procesador que almacenan temporariamente instrucciones y datos.

En la memoria principal cada posición puede ser accedida directamente por la CPU.

La memoria secundaria es de mayor tamaño y menor velocidad que la memoria principal, es usada para almacenar programas y archivos de datos que no son continuamente requeridos por la CPU.

La información en memoria secundaria es accedida a través de programas especiales que transfieren la información requerida a la memoria principal.

2.1.3. - Clasificación por la forma de ubicar la información

Las memorias de acceso por dirección operan de la siguiente manera:

La dirección de una posición requerida es transferida desde la CPU mediante el bus de dirección al registro de dirección de memoria, dicha dirección es luego procesada por el decodificador de direcciones, el cual seleccio-

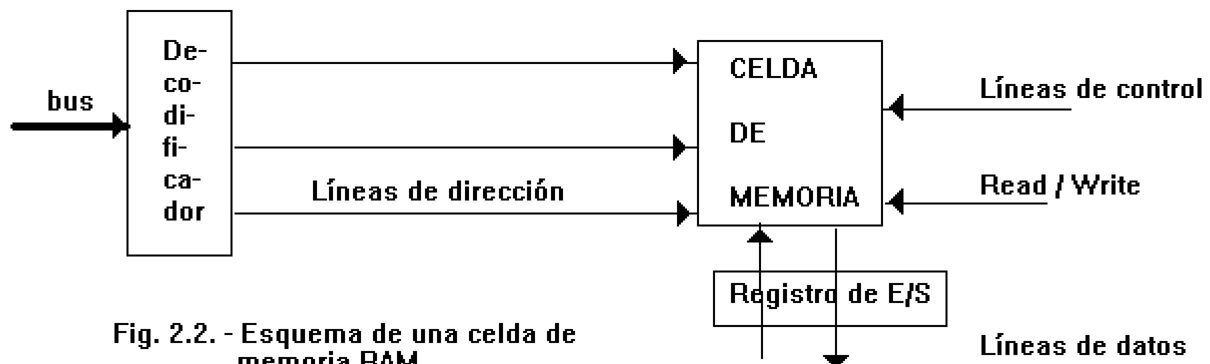


Fig. 2.2. - Esquema de una celda de memoria RAM

na la posición requerida en la memoria.

Las líneas de control de selección de Read o Write especifican el tipo de operación a realizarse. Si un Read es seleccionado, el contenido de la celda es transferido al registro de datos de salida. Si un Write es seleccionado, la palabra a ser escrita que ya se encuentra en el registro de datos de entrada de memoria es transferida a la celda seleccionada.

En las memorias asociativas o direccionables por contenido, el dato almacenado se identifica para su acceso por el contenido, en lugar de identificarse por su dirección. Cada celda de este tipo de memorias tiene capacidad de almacenar información y además circuitos lógicos para equiparar o comparar el contenido almacenado con un argumento externo.

2.1.4. - **Clasificación en base al espacio de direccionamiento**

El espacio de direccionamiento de los programas puede estar contenido en su totalidad en memoria real, o en su defecto puede estar parte en memoria real y parte almacenado en archivos de trabajo sobre algún periférico, que llamamos memoria virtual.

Estos aspectos serán tratados en profundidad en el capítulo de Administración de Memoria.

2.1.5. - **Clasificación en base a la capacidad de modificación de la información almacenada**

Las memorias RAM (Random Access Memory), también conocidas como memorias RWM (Read/Write Memory) son utilizadas para almacenar datos, variables y resultados intermedios que necesitan actualizarse.

Las memorias ROM (Read Only Memory) son utilizadas para almacenar programas y tablas de constantes que no cambian el valor una vez que han sido cargadas en la memoria del computador.

Las memorias PROM (Programmable Read Only Memory) son memorias que pueden ser programadas fuera de línea, es decir, son memorias ROM cuyo contenido puede ser programado fuera del sistema.

Las memorias EPROM (Erasable Programmable Read Only Memory) se diferencian de las anteriores en que además pueden ser reprogramadas, alterándose la programación original insertada en la PROM.

Tanto las memorias RAM, ROM, PROM y EPROM figuran normalmente en la bibliografía como memorias de acceso aleatorio (RAM).

2.1.6.- **Clasificación en base a la perdurabilidad del dato almacenado**

La noción de perdurabilidad de dato almacenado está vinculada a los procesos físicos relacionados con el almacenamiento de la información que en ocasiones suele ser inestable, de ese modo la información puede perderse luego de transcurrido determinado período, a menos que se tomen las acciones apropiadas.

Una memoria cuyo contenido puede ser destruido por una falla de energía se denomina volátil. A esta categoría pertenecen las memorias de semiconductores. Las memorias magnéticas (discos, cintas, etc) no son volátiles.

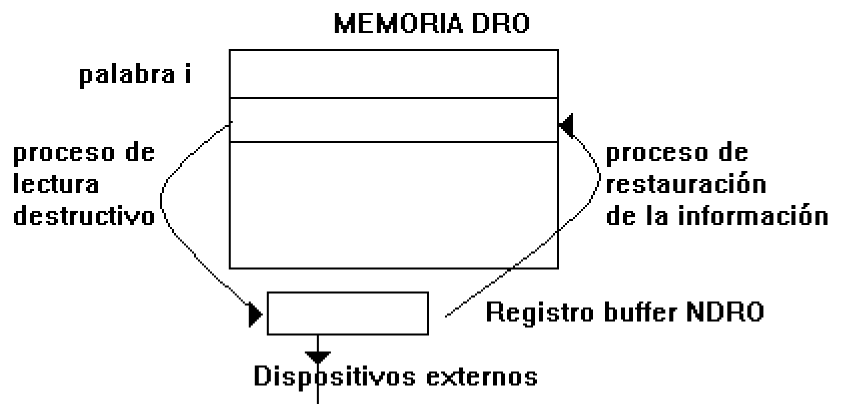


Fig. 2.3. - Ejemplo de lectura en una memoria DRO

Las memorias en las cuales el proceso de lectura altera el contenido de las mismas (por ejemplo, destruyendo la información) se denominan DRO (Destructive Read Out). Un ejemplo de éstas son las memorias con tecnología ferromagnética (núcleos).

Las memorias en las cuales el proceso de lectura no afecta el contenido se denominan NDRO (Non Destructive Read Out). En las memorias DRO cada operación de lectura debe ser seguida por una operación de regrabación que restaura el estado original de la memoria. Esta restauración es llevada a cabo automáticamente usando un registro buffer. La palabra de memoria deseada se transfiere a dicho registro buffer (memoria NDRO) y desde ese registro buffer se transfiere a los dispositivos externos, el contenido de ese registro buffer es automáticamente grabado dentro de la palabra de memoria original.

Ciertos tipos de memoria tienen la propiedad de tender a cambiar su estado con el transcurso del tiempo. Por ejemplo, las primitivas memorias con tecnología MOS (Metal Oxide Semiconductor) tendían a cambiar su estado por la pérdida de carga eléctrica en sus capacitores pasado cierto tiempo. Esto requería un proceso de restauración denominado refreshing. Las memorias que requieren este proceso periódico de restauración se denominan memorias dinámicas (Dynamic Storages). En oposición, las memorias que no requieren restauración se llaman estáticas.

El proceso de restauración en memorias dinámicas puede llevarse a cabo por un camino similar al de las memorias DRO, pero la diferencia estriba en que en estas últimas la restauración se efectúa frente a un proceso de lectura, mientras que en las memorias dinámicas el proceso de restauración es sistemático. Esto significa, en el caso de las memorias dinámicas que transcurrido un cierto período de tiempo (generalmente pequeño) se produce una interrupción al procesamiento que realiza la CPU para realizar este proceso de restauración. Tal proceso se denomina usualmente "Proceso de Refreshing".

2.1.7. - **Clasificación en base al tipo de tecnología**

En cuanto al tipo de tecnología incluiremos solamente un cuadro genérico (Ver Tabla 2.4). El mismo no cuenta con los últimos avances tecnológicos, pero hasta este punto es bastante ejemplificador en cuanto a las grandes diferencias de velocidades que se aprecian entre aquellas que corresponden a medios mecánicos y las que implican tecnología de semiconductores.

Tecnología	Tiempo de acceso	Modo de acceso	Capacidad de modific.	Perdurabilidad	Medio de almacen. físico
Semiconductor Bipolar	30-100 ns/palabra	Aleatorio	R/W	NDRO/ Volátil	Electrón.
Semiconductor Oxido Metálico	0.25-1 μs/2-32 pal	Aleatorio	R/W	DRO/NDRO volátil	Electrón.
Núcleo/Ferromagnética	5-10 μs/2-32 pal.	Aleatorio	R/W	DRO/ No volátil	Magnético
Discos y Tambores magnéticos	5-75 ms/4K	Semialeatorio	R/W	NDRO/ No volátil	Magnético
Cintas magnéticas	1-5 s/1-16 K	Serial	R/W	NDRO/ No volátil	Magnético
Tarjetas perforadas, cintas de papel perforado	1 s/ tarjeta	Serial	Read	NDRO/ No volátil	Mécanico

TABLA 2.4.

2.2. - **RAM (RANDOM ACCESS MEMORY)**

Un diagrama en bloque de una memoria RAM puede verse en la figura 2.5.

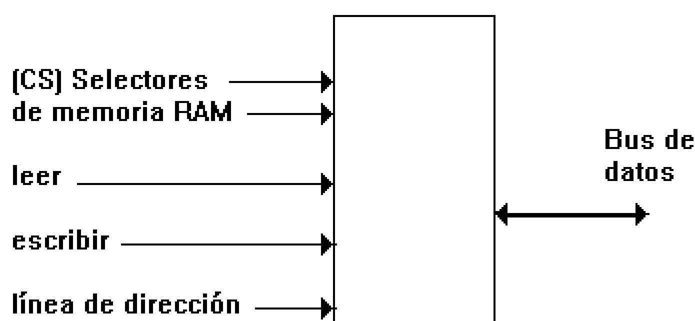


Fig. 2.5. - Una celda de memoria RAM.

Si la capacidad de memoria es de 128 palabras de 8 bits cada una se requiere una línea de dirección de 7 bits.

Los selectores (CS) se usan para seleccionar una de las memorias RAM. En realidad, una memoria RAM se compone de unidades RAM más pequeñas conectadas entre sí, y que se seleccionan mediante estas líneas de selección, es decir, de estar activas las líneas de selección de una de las memorias RAM entonces ésa es la elegida y se extrae o coloca información en dicha RAM específicamente en la posición indicada por la línea de dirección dentro de esa RAM.

Las líneas de leer/escribir pueden ser combinadas en una sola, y se usan para especificar la operación que se realizará sobre la memoria.

El funcionamiento de la RAM puede sintetizarse de la siguiente manera: si la entrada de selección de la RAM correspondiente no está activada, o si está activada pero las entradas de leer/escribir no están activadas, la memoria RAM está inhibida. Cuando la entrada de selección está activada, y la entrada de escribir está habilitada, la memoria RAM almacena un byte obtenido del bus de datos en la localización especificada por la línea de dirección. Si la entrada leer está habilitada, el contenido del byte indicado por la línea de dirección es ubicado en el bus de datos.

2.3. - **ROM (READ ONLY MEMORY)**

Una ROM está organizada internamente en una forma similar a la RAM, pero dado que la ROM solo puede leerse, el bus de datos está en modo salida. Para una ROM del mismo tamaño (en cuanto a dimensiones físicas) de una RAM es posible tener más bits, pues las celdas binarias internas de la ROM ocupan menos espacio que en la RAM ya que las dos líneas leer/escribir de la RAM no existen en la ROM (nótese que al no existir estas líneas se ahorra indudablemente el espacio dentro del circuito impreso que las mismas ocuparían y por ende la superficie libre aumenta), y pueden ser utilizadas para ampliar las líneas de dirección.

Así, una RAM que tenga una dirección de 7 bits es equivalente a una ROM de 512 bytes (9 bits). El funcionamiento de la ROM es similar al de la RAM, pero como en la ROM no hay necesidad de control de lectura/escritura, cuando una ROM es seleccionada por los selectores (CS), el byte indicado en la línea de dirección aparece en el bus de datos.

2.4. - **MEMORIA VIRTUAL**

Para entender un sistema con memoria virtual debemos distinguir los conceptos de espacio de direccionamiento lógico y espacio de direccionamiento físico. Un espacio de direccionamiento lógico es el conjunto de direcciones que aparece en un programa; el conjunto de las direcciones de memoria es el espacio de direccionamiento físico. El espacio lógico puede ser mayor que el espacio físico. Durante la ejecución de un programa, cada dirección lógica es traducida a una dirección física, este mecanismo es conocido como mecanismo de traducción (mapeo) de direcciones. La figura 2.6 muestra las componentes principales de un sistema con memoria virtual y sus conexiones lógicas.

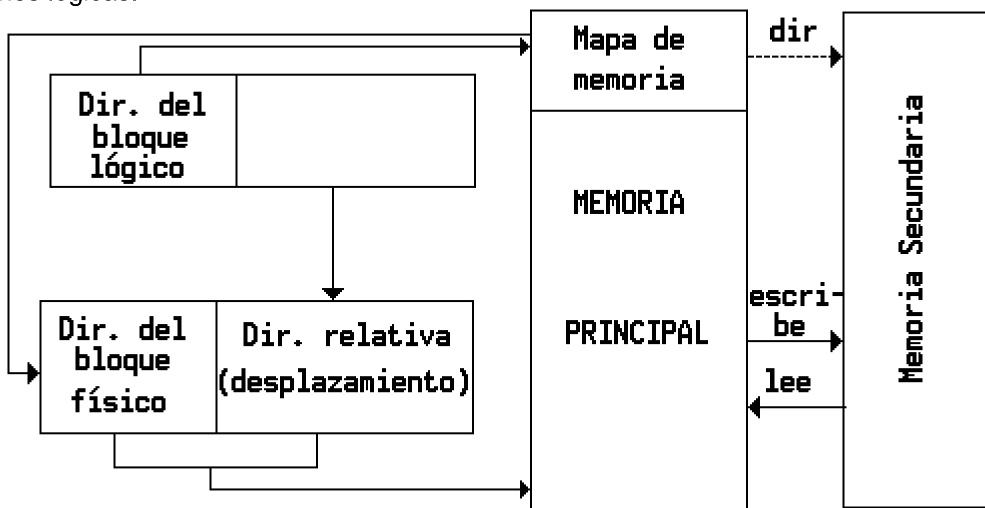


Fig. 2.6. - Componentes de un sistema con memoria virtual.

En sistemas con múltiples procesadores y memoria virtual, el mecanismo de traducción de direcciones es provisto para cada procesador.

Asumamos que el espacio de direcciones generado por el programa J en ejecución sobre un procesador es $V_j = \{0,...,n-1\}$ que consiste de n identificadores únicos. Asumamos que el espacio de memoria asignado al programa J tiene m posiciones, $M = \{0,...,m-1\}$, donde cada posición es la identificación de una única dirección de memoria. Como el espacio de memoria asignado puede variar con la ejecución del programa, m está en función del tiempo. Dicha función se define:

$f_j(x,t) =$ y si al instante t, x está en memoria.

$f_j(x,t) = \varnothing$ si al instante t, x no está en memoria.

Cuando $f_j(x,t)$ = vacío se produce una interrupción por falta de direccionamiento que provoca que las rutinas que manejan esas interrupciones requieran el ítem desde el próximo nivel de memoria (si la falta ocurre en memoria principal, se requerirá en memoria secundaria; si la falta ocurre en memoria cache, se requerirá en memoria principal).

2.4.1. - **Implementación del mapa de direcciones**

La función de traducción f puede ser implementada de diversas formas. La implementación más simple es el mapeo directo, que consiste de una tabla de n entradas donde cada entrada contiene la dirección física de memoria y (si f(x) = y), o un carácter nulo. Esta implementación requiere de accesos a memoria adicionales para llegar a la tabla de mapeo, además, las tablas de mapeo pueden ser exageradamente grandes (n posiciones), y la mayoría de sus entradas contendrían caracteres nulos (n-m entradas nulas).

Otra implementación es usar memorias asociativas para efectuar una traducción/mapeo asociativo. Este tipo de memorias asociativas se las conoce como Translation Lookaside Buffer (TLB). La traducción de dirección virtual a dirección real se realiza a través de la búsqueda por contenido (en la TLB se encuentran los pares dirección virtual-dirección física que fueron referenciados en el último lapso).

2.4.2. - **Organización de direcciones en memoria virtual**

Hay 3 métodos de organizar las direcciones en un sistema de Memoria Virtual :
1) Paginado (por demanda): organiza el espacio de direcciones en bloques de tamaño fijo llamados páginas.
2) Segmentación: organiza el espacio de los nombres (identificaciones de posiciones lógicas del programa) en bloques de tamaño arbitrario llamados segmentos.
3) Segmentación con paginado: combina los dos métodos anteriores.

2.4.3. - **Traducción de una dirección virtual a real**

La traducción de una dirección virtual a real puede sintetizarse como se ve en la Fig. 2.7.
Existe un Registro base (PTBR) que apunta a la dirección de comienzo de la tabla de páginas que le corresponde a ese proceso. Con dicha información es que se ingresa originalmente a dicha tabla.
Con el número de página que se desea acceder se utiliza la tabla de distribución de páginas para mapear en qué bloque físico de la memoria real se encuentra dicha página cargada.
Luego reemplazando el número de página por el bloque físico correspondiente y utilizando el desplazamiento provisto en la dirección virtual original se obtiene la dirección exacta de la memoria real a la que se desea acceder.

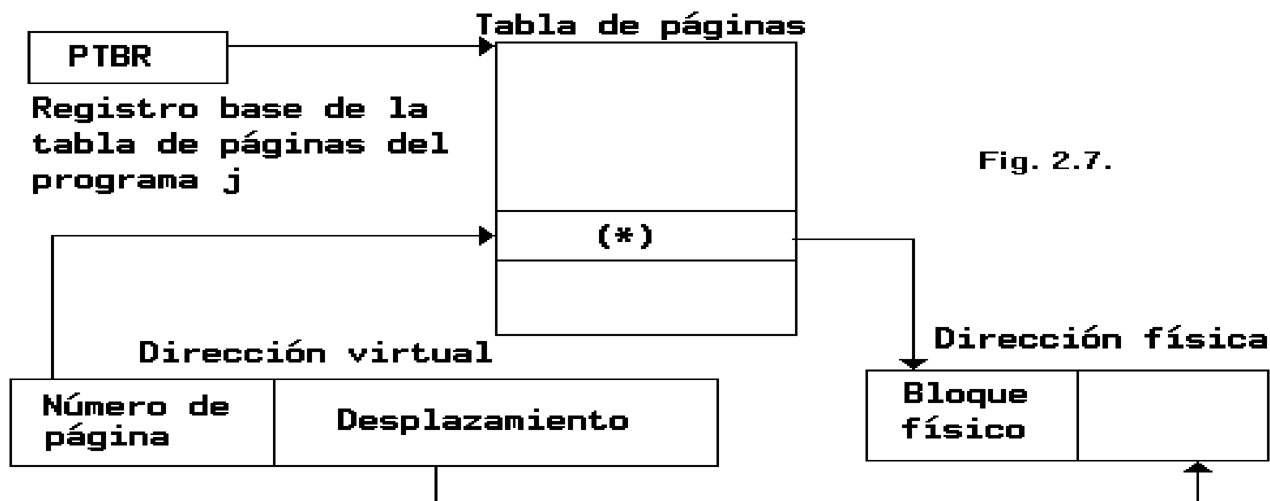


Fig. 2.7.

(*) Contiene información sobre:
- Código de acceso permitido (Read/Write/Execute).
- Bit de validez, indica si la página existe o es nula.
- Bit de página activa: indica si la página está en memoria principal o no.
- Dependiendo de si la página está activa o no:
  - i)- Dirección de la página en memoria física.
  - ii)- Dirección de la página en almacenamiento secundario.

2.5. - **MEMORIAS DE ALTA VELOCIDAD**

Como se ha visto anteriormente, el empleo de memorias de alta velocidad tiende a compensar la diferencia de bandwidth entre memoria y CPU. Entre los distintos métodos existen, a saber:
- Utilizar memorias de tecnologías rápidas (bipolar). Lamentablemente este método es muy costoso.
- Usar palabras de memoria de gran tamaño.
- Acceder a más de una palabra durante un ciclo de memoria (memorias interleaved).
- Insertar una memoria rápida (cache) entre la CPU y la memoria principal.

Clasificación por	Tipos
Método de acceso	Aleatorias Semialeatorias Secuenciales
Velocidad / Tiempo de acceso	Memoria interna del proc. (cache) Memoria Principal Memoria Secundaria
Según la forma de ubicar la información	Acceso por dirección - RAM Acceso por contenido - CAM
Según el espacio de direccionamiento	Memoria Real Memoria Virtual
Según la capacidad de modificación de la información almacenada	RAM ROM PROM EPROM
Según la perdurabilidad del dato almacenado	DRO (Destructive Read Out) / NDRO Dynamic Storage / Static Volátiles / No volátiles (discos)

TABLA 2.8. - Cuadro resumen de las clasificaciones.

2.5.1. - MEMORIAS INTERLEAVED

Este tipo de memorias es generalmente utilizada en entornos de multiprocesamiento, donde la memoria principal es compartida por varios procesadores. La memoria principal es particionada en módulos separados, o

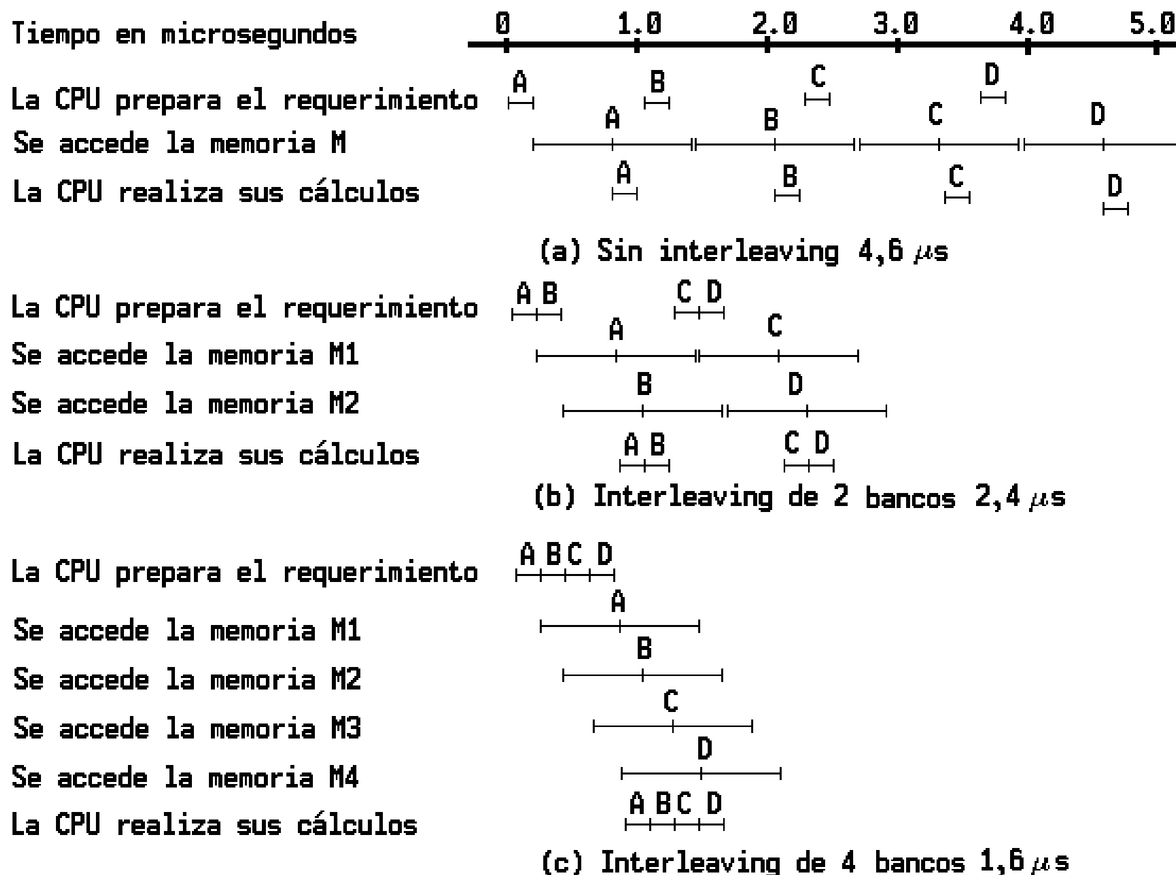


Fig. 2.9. Ejemplos de tiempos de memorias interleaved.

Bancos de Memoria, para permitir accesos simultáneos independientemente en cada módulo. Esta organización permite que una o más palabras puedan ser accedidas en cada ciclo de memoria.

Debe tenerse bien presente que si bien dos módulos de memoria pueden estar siendo accedidos simultáneamente, si existe solo un bus de datos para acceder a memoria la transferencia en sí de lo accedido debe realizarse secuencialmente.

La diferencia entre ciclo de memoria y tiempo de acceso de una memoria estriba en que el tiempo de acceso es el tiempo que le lleva a la memoria enviar o recibir una palabra (o la unidad de transferencia que corresponda) a la CPU, en tanto que el ciclo de memoria es el tiempo que transcurre desde que se hace una referencia a memoria y la misma está nuevamente disponible para poder ser accedida.

Este último tiempo que usualmente es bastante mayor al anterior proviene del hecho de que las memorias (por ser circuitos capacitores) requieren del transcurso de un cierto período de tiempo antes de poder ser nuevamente referenciadas.

Veamos un ejemplo a este respecto.

Sea una memoria cuyo tiempo de acceso es del orden de 0,6 microsegundos y cuyo ciclo es de 1,2 microsegundos, y una CPU que requiere de 0,2 microsegundos para preparar el acceso a memoria y de 0,2 microsegundos más para poder procesar la información obtenida.

En estas condiciones (Ver Fig. 2.9) acceder 4 veces a la memoria si no se utiliza la técnica de interleaving consumirá 4,6 microsegundos. En tanto que si se utiliza una memoria interleaved de 2 bancos se tardará 2,4 microsegundos y si fuera de 4 bancos el tiempo total se reduce a 1,6 microsegundos.

Cada módulo de una memoria interleaved tiene su circuito de direccionamiento. Diferentes procesadores que referencian a distintos módulos pueden acceder simultáneamente a memoria. Para que esta organización funcione eficientemente, las referencias generadas por los distintos procesadores deben ser distribuidas entre los distintos módulos, dado que dos o más referencias sobre el mismo módulo no pueden ser llevadas a cabo simultáneamente.

2.5.1.1. - **INTERCALACIÓN DE DIRECCIONES**

Si conocemos que un procesador va a referenciar a K posibles palabras de memoria ($P_0,...,P_{k-l}$), que pueden ser instrucciones dentro de un programa, y que las mismas serán asignadas a K direcciones consecutivas del direccionamiento físico ($A_0,...,A_{k-1}$), se pueden aplicar las siguientes reglas de intercalación:

1) Asignar la dirección A_i al módulo M_j si j=i [módulo m], siendo m la cantidad de módulos en que fue dividida la memoria. Es conveniente definir el número de módulos m como una potencia de 2, es decir $m = 2^p$; luego los p bits menos significativos de la dirección identifican el módulo en el cual se encuentra esa dirección. (Ver figura 2.10.)

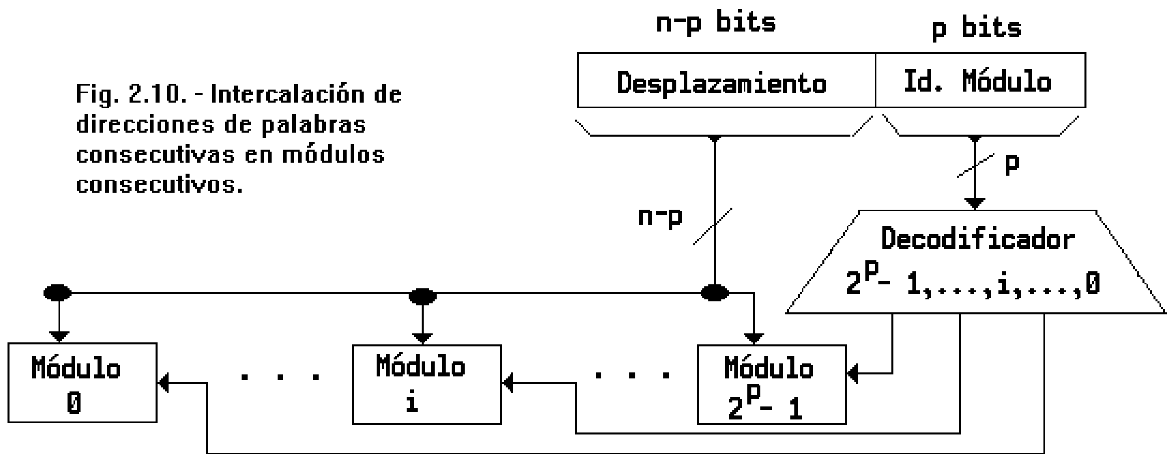


Fig. 2.10. - Intercalación de direcciones de palabras consecutivas en módulos consecutivos.

Este método de intercalación distribuye direcciones consecutivas en módulos consecutivos.

2) Asignar los bits más significativos para seleccionar el módulo, los restantes bits indican el desplazamiento dentro del módulo (Ver figura 2.11). Este esquema facilita la expansión de memoria por la adición de nuevos módulos, pero dado que direcciones contiguas están dentro del mismo módulo, puede causar conflictos en accesos simultáneos con procesadores pipeline, array processors, etc.

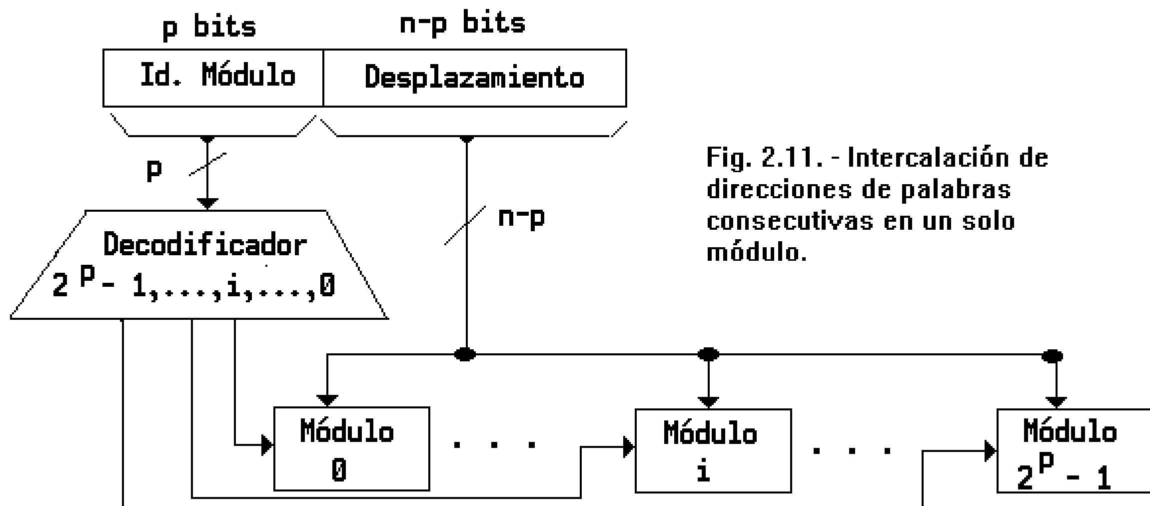


Fig. 2.11. - Intercalación de direcciones de palabras consecutivas en un solo módulo.

Esta técnica de interleaving es usada frecuentemente para incrementar la velocidad con la cual se levantan instrucciones de memoria.

La eficiencia con la cual es usado un sistema de interleaving de memoria es dependiente del orden en que fueron generadas las direcciones de memoria. Este orden es determinado por los programas que serán ejecutados. Si dos o más direcciones requieren accesos simultáneos al mismo módulo, se está en presencia de una **interferencia o contención de memoria**; los accesos a memoria en cuestión no pueden ser ejecutados simultáneamente.

2.5.2. - MEMORIAS MULTIPUERTAS Y CONEXIONES A TRAVÉS DE LINEAS (CROSSBAR SWITCH)

La figura 2.12 muestra una organización crossbar switch en la cual hay un camino (path) disponible para cada unidad de memoria:

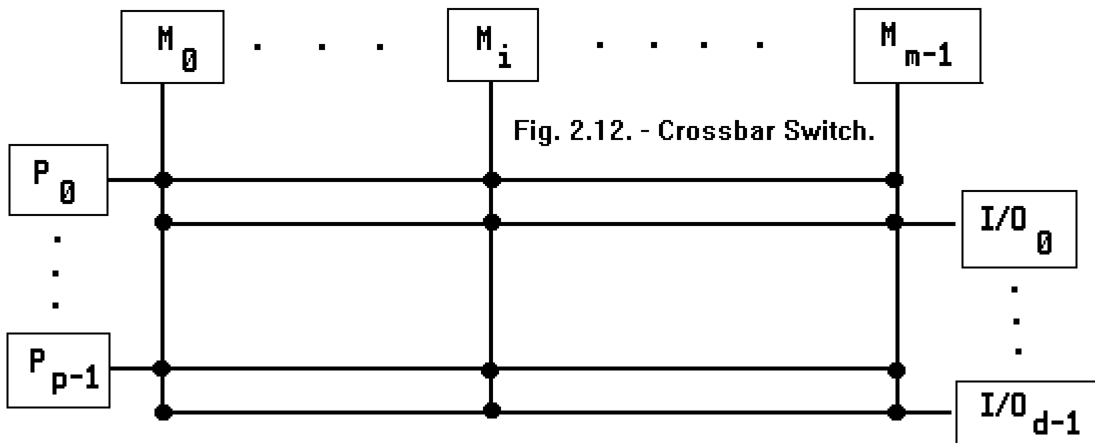


Fig. 2.12. - Crossbar Switch.

El crossbar switch brinda una conexión completa con los módulos de memoria porque existe un "bus" asociado con cada módulo de memoria. El máximo número de transferencias que pueden tomar lugar

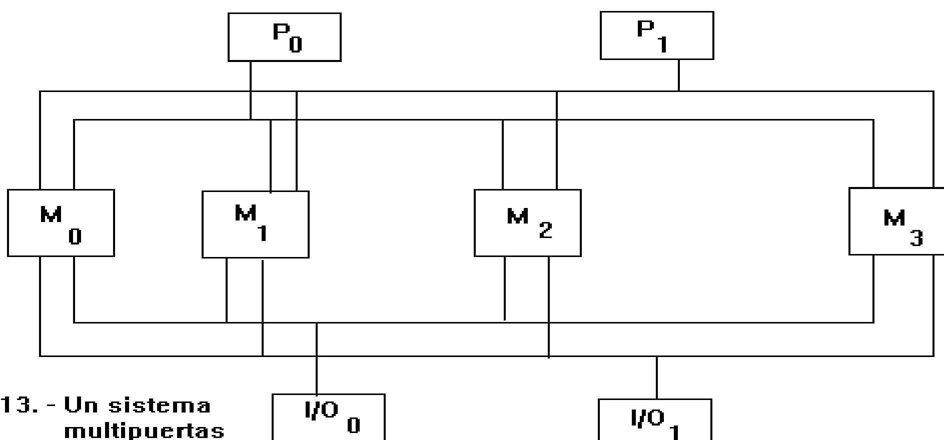


Fig. 2.13. - Un sistema multipuertas

simultáneamente está limitada por el número de módulos de memoria y la relación bandwidth-velocidad de los buses, y no por la cantidad de caminos (paths) disponibles.

Una característica importante de este sistema es la simplicidad de switchear (conmutar) unidades funcionales (es decir alterar en forma dinámica los caminos habilitados) y soportar transferencias simultáneas para todas las unidades de memoria.

Un problema que debe resolver esta organización es el arribo de múltiples requerimientos de acceso al mismo módulo de memoria en un ciclo de memoria. Este conflicto requiere del manejo de prioridades predeterminadas. Si el control y el manejo de la lógica de prioridades, que está distribuida a través de la matriz crossbar switch, se la incluye en las interfases de los módulos de memoria; el resultado es un sistema de memoria multipuerta.

La figura 2.13 muestra un sistema multipuertas. Nótese que a diferencia del gráfico del crossbar antes mencionado en este caso existen diferentes puertas de ingreso a cada elemento funcional y el método que se utiliza para resolver los conflictos de acceso a memoria es asignar prioridades a cada una de dichas puerta de acceso.

2.5.3. - **MEMORIAS ASOCIATIVAS (CAM)**

Las memorias CAM (Content Addressable Memory) son memorias direccionables por contenido, utilizadas en aplicaciones donde el procesamiento de datos requiere una búsqueda de ítems almacenados en memoria.

El dato almacenado se identifica para su acceso por el contenido, en lugar de su dirección. Debido a su organización, este tipo de memorias es accedida simultáneamente en paralelo. Las búsquedas pueden hacerse por la palabra entera o por un campo específico dentro de la palabra. Una memoria asociativa es mucho más cara que una memoria RAM. Cada celda tiene capacidad de almacenamiento y circuitos lógicos para equiparar o comparar el contenido con un argumento externo.

2.5.3.1. - **Organización del Hardware**

Una memoria de m palabras de n bits por palabra requiere un registro de equiparamiento de m bits, uno para cada una de las palabras de memoria. Cada palabra de memoria es comparada en paralelo con la clave de búsqueda que se encuentra en el registro de argumento.

2.5.3.2. **Operación de lectura**

Existe un registro argumento que contiene el dato que se desea ubicar. Además existe un registro clave cuyo fin es enmascarar parte de los bits del argumento a fin de proveer una búsqueda sobre una cantidad menor de bits de las palabras de la memoria asociativa.

Las palabras que equiparan todos sus bits con los de la clave de búsqueda activan el bit asociado a dicha palabra en el registro de equiparamiento. Después del proceso de equiparamiento, todos los bits del registro de equiparamiento que fueron activados indican el hecho de que sus palabras asociadas coinciden con el argumento (en los bits no enmascarados). La lectura se logra por un acceso secuencial a la memoria para todas aquellas palabras cuyos bits correspondientes en el registro de equiparamiento han sido activados.

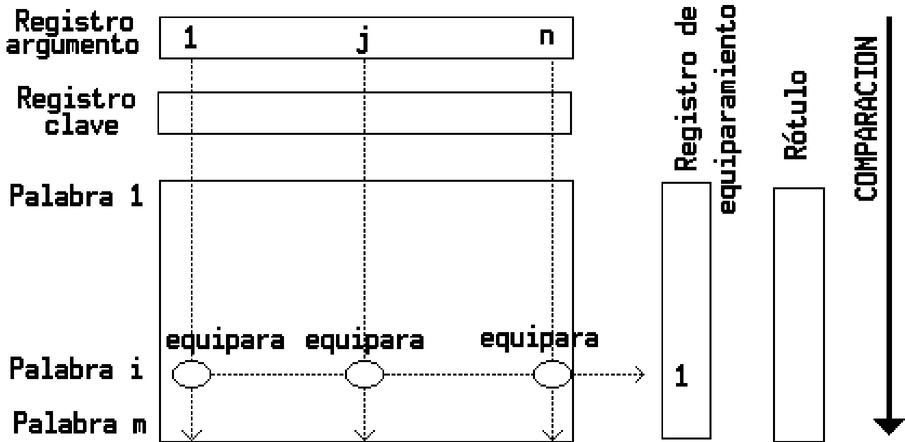


Fig. 2.14. - Lectura de una memoria CAM.

La organización interna de una celda cualquiera C_{ij} consta de un flip-flop de almacenamiento (celda binaria capaz de almacenar un bit de información) y los circuitos para leer, escribir y equiparar. (El circuito para equiparar esta constituido en base a una función booleana con inversores de bits, compuertas AND y OR).

2.5.3.3. - **Operación de escritura**

En una memoria de m palabras existe un registro especial "rótulo" de m bits que tiene por objeto llevar cuenta de las palabras activas e inactivas. Para cada palabra activa en memoria, el bit correspondiente en dicho registro se coloca en 1. Para poder almacenar una nueva palabra en memoria se recorre este registro hasta en-

contrar un bit en cero, esto da la posición de la primer palabra de memoria inactiva disponible. Después de que la nueva palabra fue almacenada se coloca su bit de rótulo en 1.

2.5.3.4 - **Tipos de memorias asociativas.**

Existen dos organizaciones diferentes de memorias asociativas:

- Organización paralela de bits: en esta organización el proceso de comparación es realizado en paralelo por palabras y en paralelo por bits. Es decir la arquitectura de estas memorias permite comparar simultáneamente por palabras completas al mismo tiempo que por franjas de bits-slices.
- Organización serial de bits: esta organización opera con un bit-slice a un tiempo a través de todas las palabras. Cada bit-slice es seleccionado por una unidad de control. Esta organización es usada en búsqueda y recuperación de información no numérica. Requiere menos hardware, pero es más lenta que la organización anterior.

2.5.4. - **MEMORIA CACHE**

Las memorias cache son buffers de alta velocidad insertados entre el procesador y la memoria principal para capturar una porción del contenido de la memoria principal que está actualmente en uso.

El éxito de las memorias cache puede ser atribuido a la propiedad de localidad de referencia. El análisis de un gran número de programas ha demostrado que las referencias a la memoria en un intervalo de tiempo tienden a estar circunscriptas dentro de un área localizada en la memoria. A este fenómeno se lo denomina localidad de referencia. Si las porciones activas del programa y los datos son colocados en una memoria rápida (más pequeña que la memoria principal), el tiempo de acceso promedio a memoria puede reducirse. El tiempo de acceso a memoria cache es menor que el tiempo de acceso a memoria principal en un factor de 5 a 10.

2.5.4.1. - **Cómo opera una Memoria Cache**

Es usual encontrar en las implementaciones que una memoria cache trabaja con una TLB asociada (Translation Lookaside Buffer), aunque puede operar sin ella.

Los guarismos para una procesador VAX-11/780 indican que la información que se necesita acceder se encuentra en la TLB más del 97 % de las veces, evitando el acceso a memoria principal.

Esto es, que el promedio de fracasos (miss) de la TLB es del orden del 3 %.

En la Fig. 2.15 puede verse un esquema genérico de funcionamiento de una cache con su TLB asociada.

El procesador utiliza los bits más significativos de la dirección virtual para obtener la información de la página referenciada desde la TLB. De encontrarse esa página en la TLB se produce una operación exitosa (TLB hit).

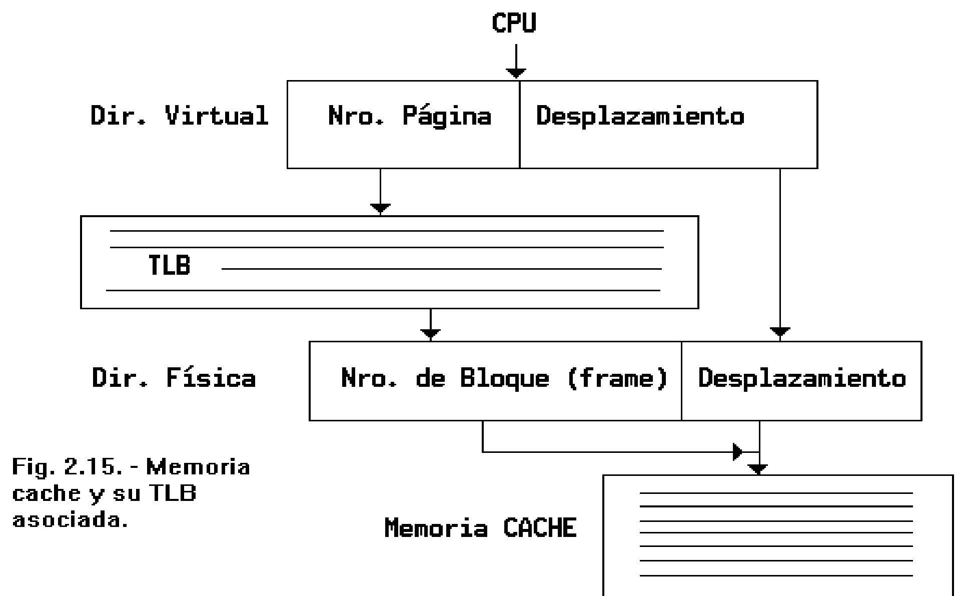


Fig. 2.15. - Memoria cache y su TLB asociada.

De la TLB se toma el número de bloque correspondiente (el bloque de memoria real que realmente ocupa dicha página), el cual se concatena con el desplazamiento original que figuraba en la dirección virtual.

Con esa información se accede entonces a la memoria cache en busca del dato requerido.

En suma la TLB contiene la información de conversión de la dirección de la página virtual al bloque de memoria real, en tanto que en la cache se hallan los pares de valores de direcciones reales y sus respectivos contenidos.

El Directorio de la Cache (CD) provee asimismo una forma más de acotar la búsqueda dentro de la misma cache.

El diagrama de flujo de la Fig. 2.16 indica cómo opera una memoria cache.

Cuando la CPU necesita acceder a la memoria, se examina la memoria cache. Si la palabra se encuentra (cache hit), la información es traída a la CPU desde la memoria cache.

Si la palabra requerida no se encuentra en memoria cache (cache miss), se accede a la memoria principal. El bloque (de 1 a 16 palabras) de la memoria principal que contiene la palabra referenciada es transferido a la memoria cache.

De esta manera, posiblemente, los datos requeridos en referencias futuras se encuentran en memoria cache.

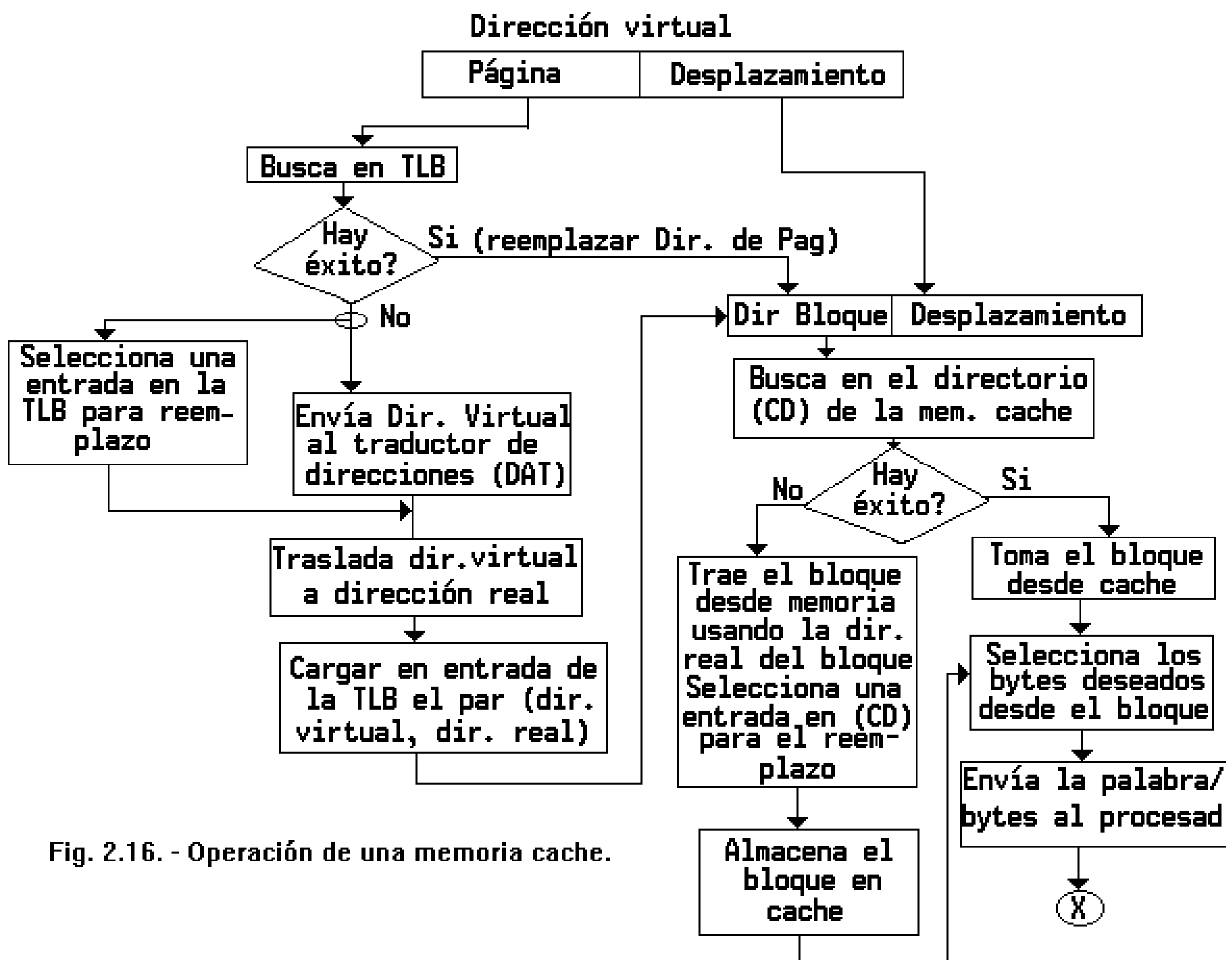


Fig. 2.16. - Operación de una memoria cache.

Las memorias cache generalmente consisten de dos partes:

- El directorio de la memoria cache (CD).
- La memoria de acceso aleatorio (RAM).

El directorio (CD) está comúnmente implementado como una memoria asociativa consistente de direcciones de bloques y bits de información adicional (bit de validez de dato, bit de protección, etc.). La ubicación de datos de la memoria principal a la memoria cache se la conoce como mapeo de datos. Existen distintos tipos de mapeo (o políticas de ubicación), a saber: directo, asociativo y asociativo de conjunto.

2.5.4.2. - **Actualización de la Memoria Cache**

Cuando la CPU encuentra una palabra en la memoria cache y la operación es de lectura, la memoria principal no está involucrada en la transferencia.

Pero si la operación es de grabación, el sistema puede actuar de dos maneras:

- El procesamiento más simple utilizado es actualizar la memoria principal cada vez que se efectúa una operación de grabación en memoria cache. Esto se denomina método de escritura directa. La ventaja del mismo es que la memoria principal siempre contiene los mismos datos que la memoria cache. Este método es importante en sistemas que poseen DMA (Direct Memory Address) puesto que asegura que los datos de la memoria principal son válidos en el instante en que los dispositivos se comunican a través del DMA.
- El segundo procedimiento es denominado método de Reescritura. En este método solamente la palabra de memoria cache es actualizada durante la operación de escritura, pero esa palabra es marcada para indicar que fue modificada y en el momento que se retire de la memoria cache (desalojo), será copiada en memoria principal.

EJERCICIOS

1) Defina qué es :

- memoria serial
- memoria semialeatoria
- memoria interna
- memoria principal
- memoria secundaria
- memoria asociativa
- memoria real
- memoria virtual
- memoria RAM
- memoria ROM
- memoria PROM
- memoria EPROM
- memoria DRO
- memoria NDRO
- memorias dinámicas
- memorias estáticas
- memoria volátil

2) Dibuje esquemáticamente cómo es una celda de una memoria RAM. En qué se diferencia con una memoria ROM ? Indíquelo expresamente en su dibujo.
3) Qué es memoria interleaved y cómo funciona ?
4) En qué esquema de memoria se puede producir lo que se denomina **contención de memoria** ?
5) Explique en qué consiste el mecanismo de refreshing y en qué tipos de memoria es necesario.
6) Explique cómo funciona el direccionamiento en una memoria de tipo asociativa (CAM).
7) Cuál es la diferencia entre ciclo de memoria y tiempo de acceso ?
8) Justifique la utilidad de las memorias interleaved.
9) Dado un programa secuencial con direccionamiento de instrucciones y datos correlativos qué tipo de Intercalación de direcciones conviene ? Justifique.
10) Explique con palabras qué es una memoria cache y cómo opera.
11) Explique en qué situación se produce un "TLB miss" ?
12) Cuáles son los dos métodos que se utilizan normalmente para asegurar la integridad de la información de memoria principal con respecto a la de memoria cache y en qué consisten ?

CAPITULO 3

SUBSISTEMAS DE ENTRADA / SALIDA

3.1 - Características de los subsistemas de E/S:

Un subsistema de E/S consiste en interfases de E/S y dispositivos periféricos. Un diagrama típico de esta arquitectura puede verse en la Fig. 3.1.

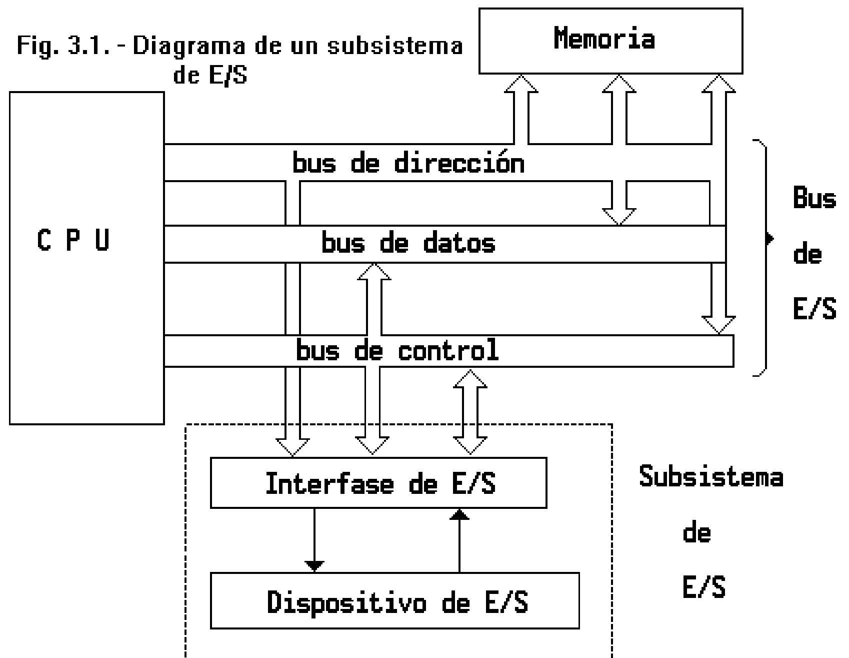


Fig. 3.1. - Diagrama de un subsistema de E/S

La interfase de E/S controla la operatoria de los dispositivos conectados a ella. Las operaciones de control (por ejemplo rebobinado, posicionamiento, etc) se arrancan mediante comandos emitidos por la CPU. El conjunto de comandos que se ejecutan para completar la transacción de E/S se denomina driver.

Las funciones de la interfase son almacenar los datos y realizar las conversiones que se le requieran. También detecta errores en la transmisión y es capaz de reiniciar la transacción en casos de error. Más aún, la interfase puede testear, arrancar y detener el dispositivo según las directivas impartidas por la CPU. En algunos casos la interfase puede consultar a la CPU si algún dispositivo está requiriendo atención urgente.

En la figura 3.2 podemos ver la clase de hardware que se encuentra en el bus de E/S.

Existen distintos tipos de comandos que circulan por el bus, a saber:

- De control: son para activar el periférico y decirle que debe hacer (por ej. rebobinar una cinta); varían según cada tipo de periférico.
- De verificación: verifican las diversas condiciones de estado en la interfase o en el periférico (por ej., una vez seleccionada la ruta la CPU puede desear verificarla para ver si existe energía (power on) o que el periférico esté en línea (on line).
- Salida de datos: Hace que la interfase responda tomando un ítem de datos del bus.
- Entrada de datos: la interfase recibe un ítem de datos del periférico y lo coloca en su propio registro separador, avisa a la CPU, la que emite el comando de entrada de datos el cual transfiere el contenido de ese registro al bus de donde es tomado por la CPU y almacenado en su registro acumulador.

Ejemplo: Salida de datos a una unidad de cinta.

El computador arranca la unidad de cinta emitiendo un comando de control. El procesador entonces monitorea el estado de la cinta por medio de comandos de verificación.
Cuando la cinta está en posición correcta, el computador emite un comando de salida de datos.

La interfase responde a la dirección y a las líneas de comando y transfiere los datos de la línea de datos del bus de E/S a su registro separador. La interfase se comunica entonces para aceptar un nuevo ítem de datos para almacenar en la cinta.

3.2 - Modos de transferencia

Los subsistemas de E/S pueden clasificarse según qué tanto esté involucrada la CPU en la transacción de E/S. Una transacción de E/S puede transferir un único bit, byte, palabra, o bloque de bytes de información entre el dispositivo de E/S y la CPU, o entre el dispositivo de E/S y la memoria principal.

3.2.1 - Transferencia de datos bajo control de programa

La arquitectura más simple de E/S es aquella en la cual el procesamiento se realiza en forma secuencial. En tales sistemas, la CPU ejecuta programas que inicializan, chequean el estado del dispositivo, realizan la transferencia de los datos, y terminan las operaciones de E/S. En este caso, las transacciones de E/S son realizadas utilizando un programa-manejador de E/S. Muchas computadoras proveen esta opción ya que requiere un hardware mínimo.

Sin embargo, la CPU permanece en un ciclo de programa testeando el dispositivo hasta el momento en que este indique su disponibilidad, lo cual trae como desventaja además de que se degrada el uso de la CPU que el tiempo requerido para transferir una unidad de información entre memoria y el dispositivo de E/S es de una magnitud varias veces superior al tiempo promedio del ciclo de instrucción. Una posible solución a esta degradación es permitir la concurrencia entre el procesamiento de la CPU y las E/S.

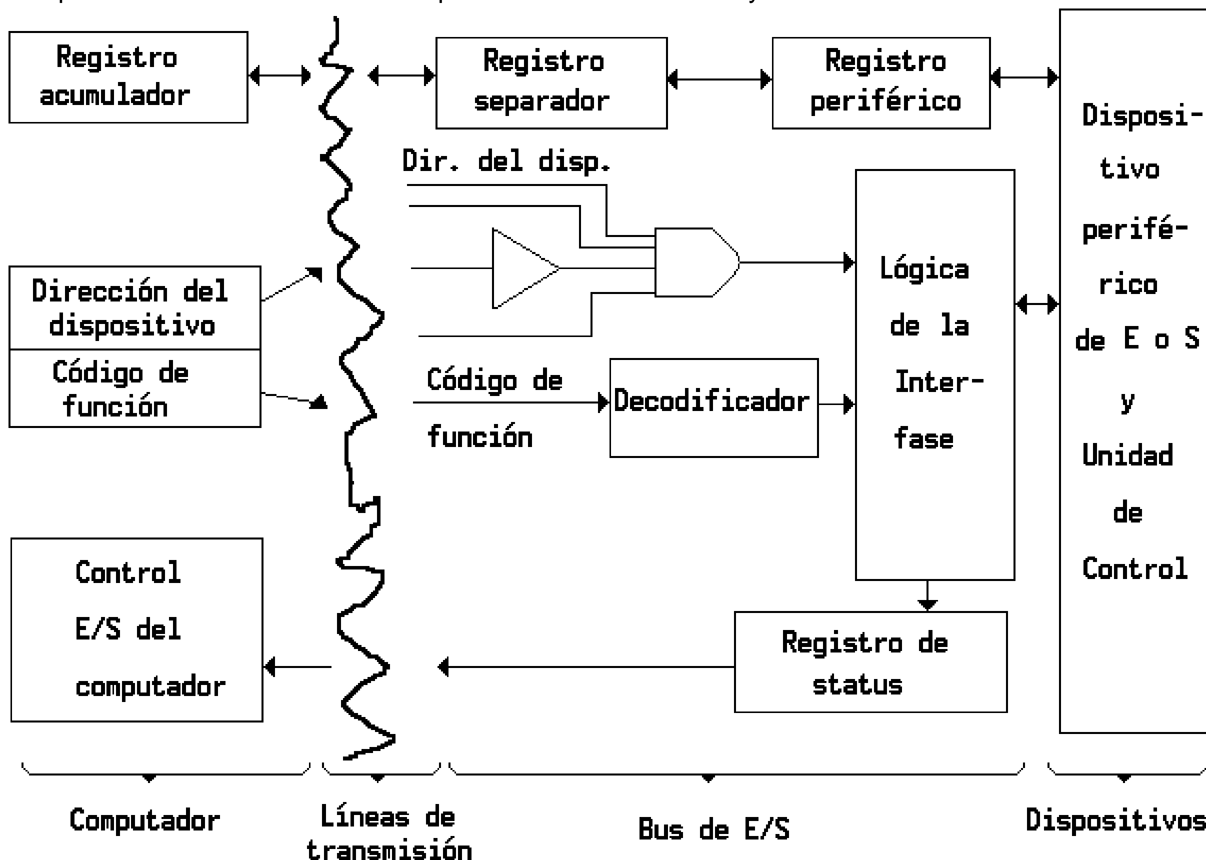


Fig. 3.2. - Hardware dentro de un bus de E/S.

3.2.2 - **Transferencia de datos con mecanismo de interrupción**

A medida que el nivel de concurrencia aumenta, la complejidad del hardware deberá crecer también para poder satisfacer los requerimientos de transferencia de datos.
Un esquema utiliza un "pseudo" programa-manejador de E/S. En este esquema la CPU inicia la transacción de E/S y reasume su tarea normal. Cuando el dispositivo tiene el dato listo, por ejemplo una operación de input, la unidad de control del dispositivo notifica a la CPU la presencia del dato en su buffer.

La CPU puede entonces atender al dispositivo para ubicar el dato y transferirlo a memoria (nótese que en este caso es la misma CPU la que coloca el dato en la memoria una vez que esta disponible desde el medio externo).

Una descripción similar puede hacerse sobre una operación de output.

La señal de notificación a que nos referimos es un requerimiento de **interrupción**. La capacidad de interrupción libera a la CPU de estar constantemente chequeando el estado del dispositivo.

Debido a que una interrupción puede producirse en forma asincrónica durante un ciclo de instrucción, muchos procesadores permiten que la instrucción se complete antes de atender la interrupción. La CPU indica su disponibilidad para atender interrupciones mediante una señal de habilitación-de-interrupción. Esta señal notifica al dispositivo si la CPU se encuentra en estado no-interrumpible. Cuando se produce la interrupción la CPU notifica al dispositivo el hecho de que toma conocimiento de dicha interrupción enviándole una señal de recibida al controlador del dispositivo.

3.2.3 - **Transferencia directa a memoria (DMA)**

El último grado de concurrencia en procesamiento de E/S puede alcanzarse si el controlador del dispositivo es lo suficientemente inteligente como para realizar la transacción de E/S entre el dispositivo y la memoria principal sin la intervención de la CPU.

Este paralelismo es muy efectivo cuando se transmite un bloque de datos. Esto requiere que el controlador sea capaz de generar una secuencia de direcciones de memoria. Sin embargo *la CPU es aún responsable de iniciar la transferencia del bloque.*

Veamos por ejemplo una secuencia típica de operaciones necesarias para transferir un bloque de datos entre un dispositivo y memoria. La CPU inicializa el buffer en la memoria principal que recibirá el bloque de datos luego de que se completa la transacción de E/S. La dirección del buffer y su tamaño son transmitidos al controlador del dispositivo, así como la dirección del bloque de datos dentro del dispositivo. Entonces la CPU ejecuta un comando especial de inicio de E/S (start I/O) que provoca que el subsistema comience a transferir.

Mientras la transferencia se produce, la CPU queda liberada para poder atender a sus cálculos lo que mejora la performance general del sistema. Cuando se completó la transferencia se notifica a la CPU.

Nótese que ya que la CPU y el controlador comparten la memoria principal, el dispositivo debe periódicamente "robar" ciclos de memoria para depositar los datos en ella.

El mecanismo de "robo" de ciclos es muy efectivo debido a que los dispositivos son muy lentos respecto de la CPU. Cuando existe conflicto entre el dispositivo y la CPU se le da prioridad al dispositivo sobre la CPU debido a que es un componente de mayor tiempo crítico.

Este tipo de esquema de transferencia de datos se denomina Acceso Directo a Memoria **(Direct Access Memory - DMA)**.

La diferencia principal entre una transferencia controlada por programa de E/S y DMA, es que *el DMA no emplea los registros de la CPU* (por ejemplo para llevar la cuenta de los bytes que se van transfiriendo).

La transferencia es hecha en la interfase de la DMA primero verificando si la unidad de memoria no está siendo utilizada por la CPU y posteriormente la DMA "roba" un ciclo de memoria para acceder una palabra de memoria.

El controlador de E/S utilizado generalmente para operaciones DMA se denomina "canal de datos de E/S". Un canal de datos es un procesador de E/S que puede manejar muchos periféricos a través de un DMA y una facilidad de interrupción.

3.2.4 - **IOP's (Input Output Processor)**

Nótese que la facilidad DMA no concede el control total de la transacción de E/S al subsistema de E/S.

El subsistema de E/S puede asumir el completo control de las transacciones de E/S si se utiliza una unidad especial llamada "procesador de E/S" (IOP).

El IOP tiene acceso directo a memoria principal y contiene una cantidad de canales independientes de E/S de datos. Puede ejecutar programas de E/S y puede realizar algunas transacciones independientes de E/S entre memoria y los dispositivos o entre dos dispositivos, sin intervención de la CPU.

Los canales proveen una vía de comunicación entre el IOP y las unidades de control de los dispositivos y los dispositivos.

El sistema de bus-IOP comunica los procesadores de E/S y los dispositivos de E/S entre sí, en tanto que el sistema de bus de memoria comunica el IOP con la memoria principal. Los canales de E/S pueden existir solos sin un IOP.

En la forma más simple, y cuando existen solos, los canales pueden ser pequeños procesadores que realizan operaciones DMA para un pequeño conjunto de dispositivos. Si el canal se incorpora en un IOP, es esencialmente un componente pasivo sin capacidad de procesamiento lógico sobre lo que le pertenece. Cuando el canal posee capacidad de procesamiento es utilizado muy a menudo como un IOP.

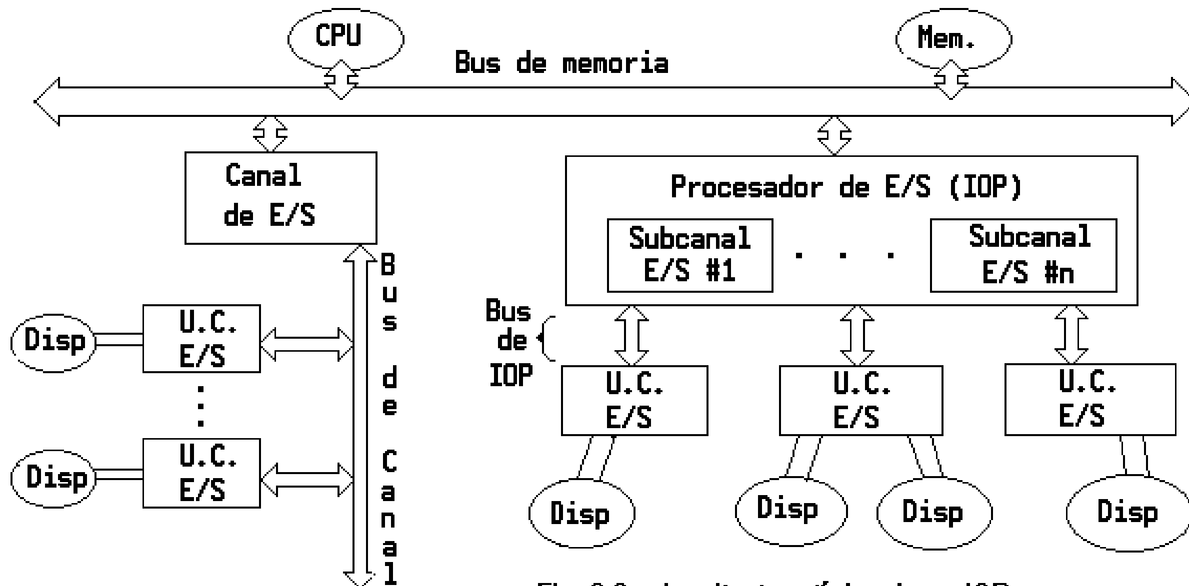


Fig. 3.3. - Arquitectura típica de un IOP.

Los canales stand-alone son utilizados en muchos mainframes, como por ejemplo la IBM 370; los IOPs se utilizan en sistemas tales como la CDC 6600 y en microcomputadoras Intel de 8 y 16 bits. En la figura 3.3 puede verse una arquitectura típica de un sistema con IOP y canales stand-alone.

Para realizar una transacción de E/S la CPU transmite una señal de start y la dirección del dispositivo al canal del cual pende dicho dispositivo. Estando la ruta libre el canal obtiene entonces la dirección de comienzo del programa de canal para poder ejecutarlo, y lo ejecuta para realizar la transacción de E/S; de otra forma el requerimiento debe ser encolado o debe notificarse a la CPU respecto de la no disponibilidad del dispositivo. En la figura 3.4 puede verse el mecanismo de comunicación entre un IOP y la CPU.

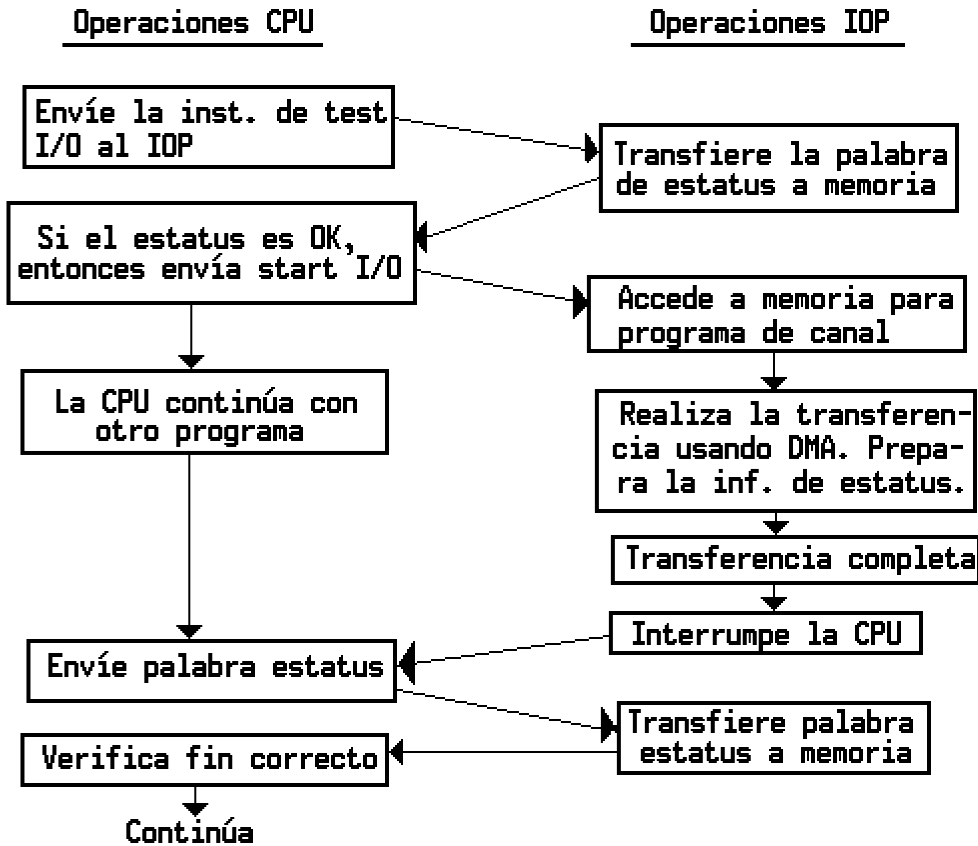


Fig. 3.4. - Mecanismo de comunicación entre CPU - IOP.

3.3 - **Canales selectores y multiplexores: su arquitectura**

Algunos canales de E/S están permanentemente conectados a un único dispositivo (es decir, se necesita un canal por cada dispositivo) mientras que otros pueden estar conectados a la vez con varios dispositivos, a los que pueden accionar uno por vez (canales selectores) o con simultaneidad (canales multiplex).

Un canal selector es un IOP diseñado para manejar una sola transacción de E/S a la vez en forma dedicada. Una vez que el dispositivo es seleccionado, el conjunto de operaciones de E/S para tal transacción se ejecuta hasta finalizar, antes de que pueda iniciarse la próxima transacción. La tasa de transmisión máxima de un canal selector es típicamente del orden de los 1 a 3 megabytes/s.

Un canal multiplexor es un IOP que puede controlar algunas transacciones diferentes de E/S concurrentemente. En este caso, las transferencias de datos son intercaladas (multiplexadas) en el tiempo íntegramente en la interfase de E/S.

Estos tipos de canales pueden dividirse en: block multiplexor o byte multiplexor. El byte multiplexor se utiliza para dispositivos lentos, y el block multiplexor se usa para dispositivos de velocidad media-alta.

El multiplexor consiste en un conjunto de subcanales cada uno de los cuales actúa como un canal selector de baja velocidad. Cada subcanal contiene un buffer, un registro de dirección de dispositivo y flags (banderas, señales) que pueden indicar el estado u operaciones de control.

Sin embargo, los subcanales comparten el control global del canal. En la modalidad multiplex el control del canal recorre cíclicamente las flags de cada subcanal. Si la señal está encendida, el subcanal es seleccionado para transferir un carácter o un bloque. Se chequea la modalidad del subcanal para determinar la dirección de la

transferencia. Cuando el carácter o bloque ha sido transferido se examina el siguiente subcanal. El block multiplexor intercala bloques, así como el byte multiplexor intercala caracteres.

Por ejemplo, supongamos que se requieren tres transacciones de E/S para obtener X, Y y Z. Supongamos que cada transacción requiere transmitir un conjunto de n caracteres (cada carácter es 1 byte) X1 a Xn; Y1 a Yn y Z1 a Zn.

Si estas transacciones son iniciadas en un canal selector la transacción aparece de esta forma:

X1,...,Xn,Y1,...,Yn,Z1,...,Zn

En un canal multiplexor de tipo byte multiplexor, con por lo menos tres subcanales, puede aparecer así:

X1,Y1,Z1,...........Xn,Yn,Zn

Si el canal es del tipo block multiplexor de k caracteres por bloque (k menor a n), la transferencia aparece de la siguiente forma:

X1,...,Xk,Y1,...,Yk,Z1,...,Zk,Xk+1,...

3.4 - **Configuraciones de E/S en un sistema con memoria cache**

Hay dos formas básicas de conectar el subsistema de E/S a una cache.

En el primero el canal de E/S puede ser conectado a la cache. De esta forma la cache es compartida por el procesador y los canales como muestra la figura 3.5.

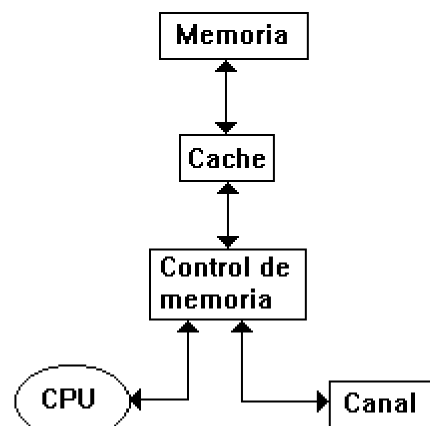


Fig. 3.5. - Canal accede a cache.

El canal compite con el procesador por accesos a la cache. Un canal de E/S es mucho más lento que el procesador, luego conectar el canal a la cache no mejora significativamente la performance de la transferencia de E/S.

En este esquema de conexión existe un incremento del tráfico entre la memoria principal y la cache, debido a la actualización de la memoria principal con la información transferida por el canal, éste debe levantar desde memoria principal las instrucciones del correspondiente programa de canal por medio de la cache, además disminuye el espacio disponible de cache a los procesos.

Una configuración alternativa es conectar el canal directamente a memoria, como se ve en la figura 3.6.

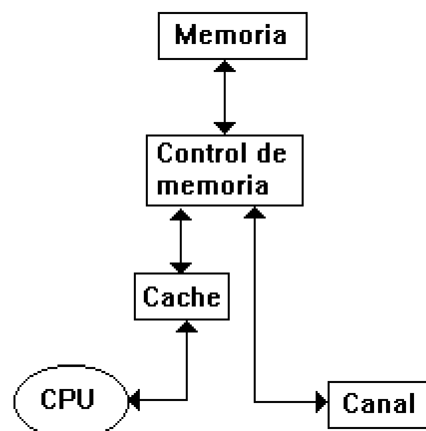


Fig. 3.6. - Canal accede a memoria.

En este caso el canal compite con la cache para acceder a memoria principal. Aún subsiste el conflicto entre la CPU y el canal para el acceso a memoria principal, pero solamente cuando la CPU detecta la falta de la información en la cache.

Existe aún un problema de coherencia entre la información en la cache y la información accedida en memoria principal por el canal.

Una forma posible de solucionar este problema es obligar al canal a verificar si la palabra de memoria que está accediendo se encuentra en memoria cache y testear el bit de cambio de esa palabra en la cache. Luego podrá determinar si es necesario una actualización del dato en memoria principal desde la memoria cache antes de recuperar la dirección deseada.

EJERCICIOS

1) Dibuje esquemáticamente como se compone un subsistema de E/S. Indique claramente qué clase de información circula por cada uno de los buses de interconexión.

2) Indique las diferencias en los siguientes métodos de transferencia de información dentro de un subsistema de E/S :

- Transferencia de datos bajo control de programa
- Transferencia de datos con mecanismo de interrupción
- Transferencia directa a memoria
- Transferencia por Procesadores de E/S

Indique claramente en cuál de estos métodos aparece por primera vez el elemento hardware **canal**.

3) Tiene sentido tener IOP's en un sistema monoprocesador ? Justifique.

4) Cómo funciona un canal byte-multiplexor ? Y un canal block-multiplexor ?

5) Cuáles son las dos formas de conectar una memoria cache en un subsistema de E/S dado ?

CAPITULO 4

PROCESAMIENTO EN PARALELO : PIPELINE.

4. - INTRODUCCIÓN A ARQUITECTURAS PARALELAS.

Muchos de los computadores antiguos y muchos de los minicomputadores contemporáneos son monoprocesadores, lo cual no debe sorprendernos ya que es la máquina más elemental para diseñar y construir.

Por otra parte, las computadoras digitales modernas de gran escala utilizan frecuentemente el procesamiento simultáneo (concurrencia) en distintos puntos del sistema, denominaremos a esta clase de computadoras Procesadores Paralelos.

Los computadores paralelos son sistemas de computadores consistentes de un conjunto centralizado de procesadores que pueden procesar simultáneamente los datos del programa.

El procesamiento paralelo se basa en la explotación de sucesos concurrentes en el proceso de cómputo.

Como apuntamos en el Capítulo 1 la concurrencia implica paralelismo, simultaneidad y pipelining.

Sucesos Paralelos ocurren en múltiples recursos durante el mismo intervalo de tiempo.

Sucesos Simultáneos ocurren en el mismo instante.

Sucesos Pipeline ocurren en lapsos superpuestos.

Se puede hablar de niveles de paralelismo, que caracterizamos de la siguiente manera:

- *Multiprogramación, Multiprocesamiento*: Estas acciones se toman a nivel de Programa o Trabajo.
- *Tarea o Procedimientos*: Acciones que se toman dentro de un mismo programa, ejecutándose procesos independientes en forma simultánea.
- *Interinstrucciones*: Acciones a nivel de instrucción, o sea, dentro de un mismo proceso o tarea se pueden ejecutar instrucciones independientes en forma simultánea.
- *Intrainstrucciones*: Acciones simultáneas que se pueden realizar para una misma instrucción, por ejemplo vectorización de operaciones escalares dentro de una instrucción compleja tipo DO, FOR, etc.

El paralelismo de un mayor nivel se obtiene por medio de algoritmos, los de menor nivel con importante actividad del hardware.

Últimamente ciertas técnicas del procesamiento distribuido son incorporadas a arquitecturas centralizadas para conseguir mayor grado de paralelismo.

El paralelismo puede obtenerse de distintas maneras, a saber:

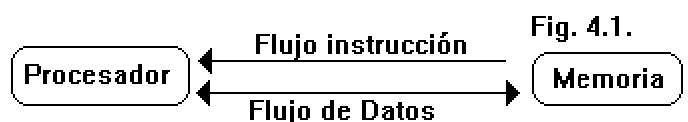
- **Multicomputadoras**: Computadoras independientes, muy a menudo una de ellas actúa como supervisor, que realizan una tarea común en una sola ubicación (una configuración muy común, aunque ciertamente limitada, es la minicomputadora como preprocesador de un computador mainframe).
- **Multiprocesadores**: Un conjunto de unidades de cómputo, cada una de las cuales tiene sus propios conjuntos de instrucciones y datos, compartiendo una misma memoria. Los computadores multiprocesadores consisten en un número n mayor o igual a 2 de procesadores que operan simultáneamente sobre una misma memoria, y están interconectados mediante canales que transmiten comandos de control y datos. Están controlados por un único Sistema Operativo.
- **Redes de computadoras**: Computadoras independientes conectadas mediante un canal de manera tal que los recursos propios disponibles en un punto de la red pueden estar disponibles para todos los miembros de la red.
- **Procesador Pipeline:** Un solo computador el cual puede realizar simultáneamente operaciones de cálculos en determinadas secciones, con diferentes estadios de completitud. Los procesadores pipeline se basan en el principio de dividir los cálculos entre una cantidad de unidades funcionales que operan simultáneamente existiendo superposición.
- **Procesador Array:** Un grupo de unidades de cómputo cada una de las cuales realiza simultáneamente la misma operación sobre diferentes conjuntos de datos. Los procesadores array operan sobre vectores. Las instrucciones del computador vectorial son ejecutadas en serie (como en los computadores clásicos) pero trabajan en forma paralela sobre vectores de datos.

4.1. - CLASIFICACIÓN DE FLYNN.

A grandes rasgos, la ejecución de una instrucción puede verse como etapas distintas, que realizan:

- Búsqueda de la instrucción,
- Decodificación de la instrucción,
- Búsqueda de los operandos, y
- Ejecución de la instrucción.

Fig. 4.1.



Las instrucciones pueden verse como un flujo de instrucciones que se desplazan de memoria al procesador y los operandos como un flujo de datos que se desplazan entre memoria y el procesador (Fig. 4.1).

Analizando el ciclo de una instrucción se puede diferenciar a un procesador en dos unidades funcionales:

La unidad de control (CU): decodifica las instrucciones y envía señales a una Unidad de Procesamiento.

La unidad de procesamiento (PU): ejecuta las instrucciones decodificadas y envía los resultados a la unidad funcional memoria.

M. J. Flynn en 1966 realizó una clasificación del paralelismo presente en un procesador basado en el número de Flujo de Instrucciones y de Datos simultáneos.

Combinando flujo de instrucciones y flujo de datos surgen 4 grupos de procesadores posibles.

En el caso de monoprocesamiento :

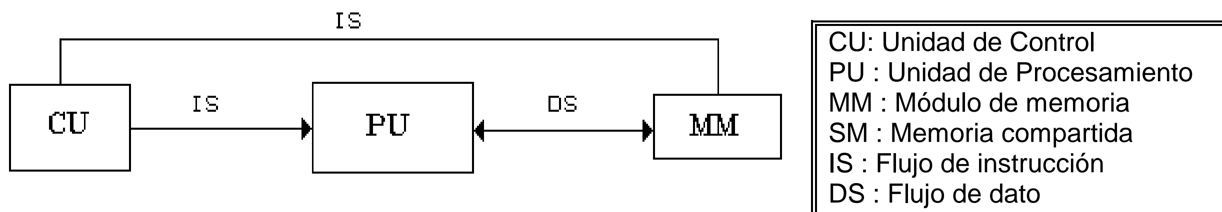


Fig.4.2. - Computador SISD.

CU: Unidad de Control
PU : Unidad de Procesamiento
MM : Módulo de memoria
SM : Memoria compartida
IS : Flujo de instrucción
DS : Flujo de dato

Tabla 4.3.

Este esquema se encuentra normalmente en la literatura como SISD (Single Instruction Single Data), o sea una sola instrucción, un solo dato a la vez.

Por ejemplo si se tuviera un programa de las siguientes características :

```
        DO 20  I=1,10
        A(I) = A(I) * 3
20      CONTINUE
```

Las diez veces que se pasa por la asignación, serán diez instrucciones, diez ejecuciones completamente independientes una de la otra.

Si se pudiera implementar el ejemplo anterior en forma paralela, en diez unidades de procesamiento simultáneas bajo el control de una única unidad de control ejecutaríamos las diez instrucciones en forma simultánea, con lo cual se ahorra mucho tiempo.

Esta idea se traduce esquemáticamente en la arquitectura que se visualiza en la Fig. 4.4.

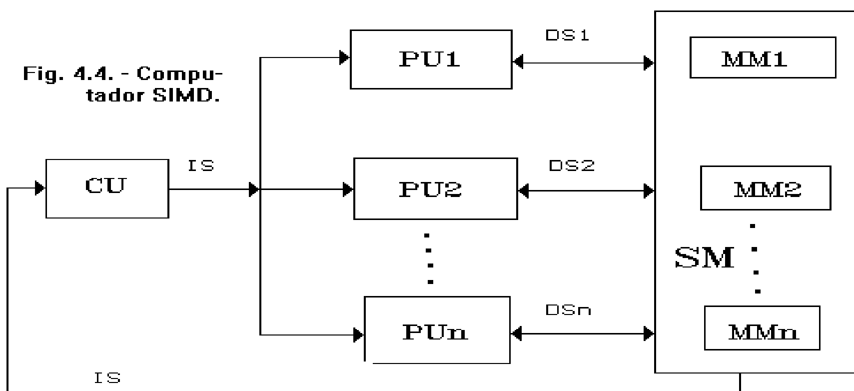


Fig. 4.4. - Computador SIMD.

O sea, tener **una** memoria, **una** unidad de control, y de dicha unidad de control tener **varias** unidades de ejecución que depositarían sus datos en memoria. A estas arquitecturas se las encuentra en la literatura como SIMD (Single Instruction Multiple Data), o sea una sola instrucción que ataca a muchos datos. Este es el esquema más básico, de lo que comúnmente se conoce como un procesador vectorial. Que es capaz de tomar un vector o matriz, y simultáneamente, hacer muchas operaciones sobre él.

Luego existe un caso distinto :

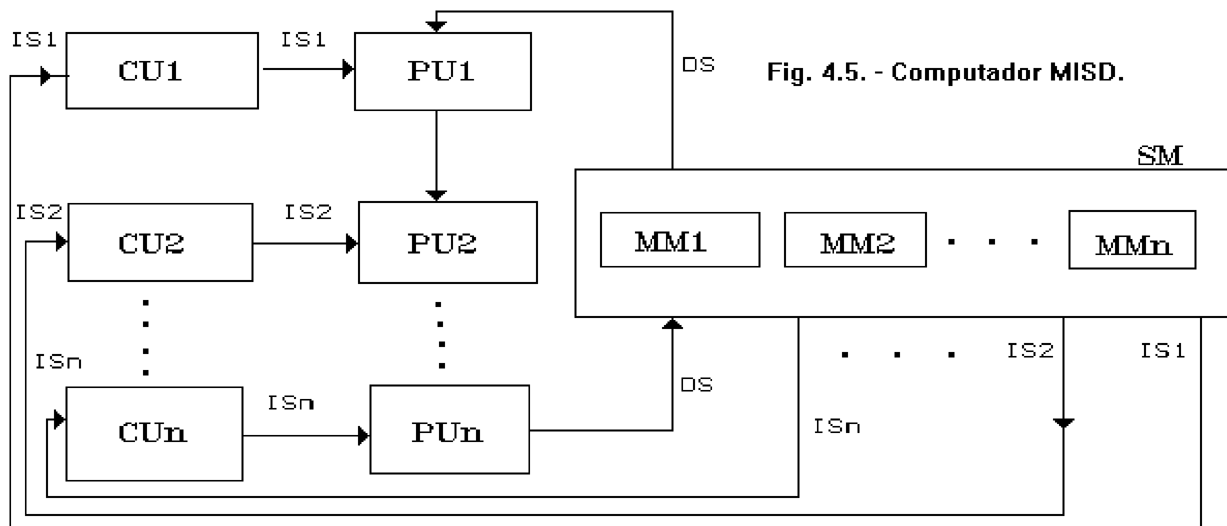


Fig. 4.5. - Computador MISD.

Aquí se tienen varias unidades de control, con varias unidades de ejecución, existen distintos flujos de instrucciones y un **único flujo de datos**. O sea se tienen múltiples instrucciones con un solo dato, MISD (Multiple Instruction Single Data).

Es un esquema teórico, en realidad no existe ningún procesador que esté implementado.
Y por último se tiene :

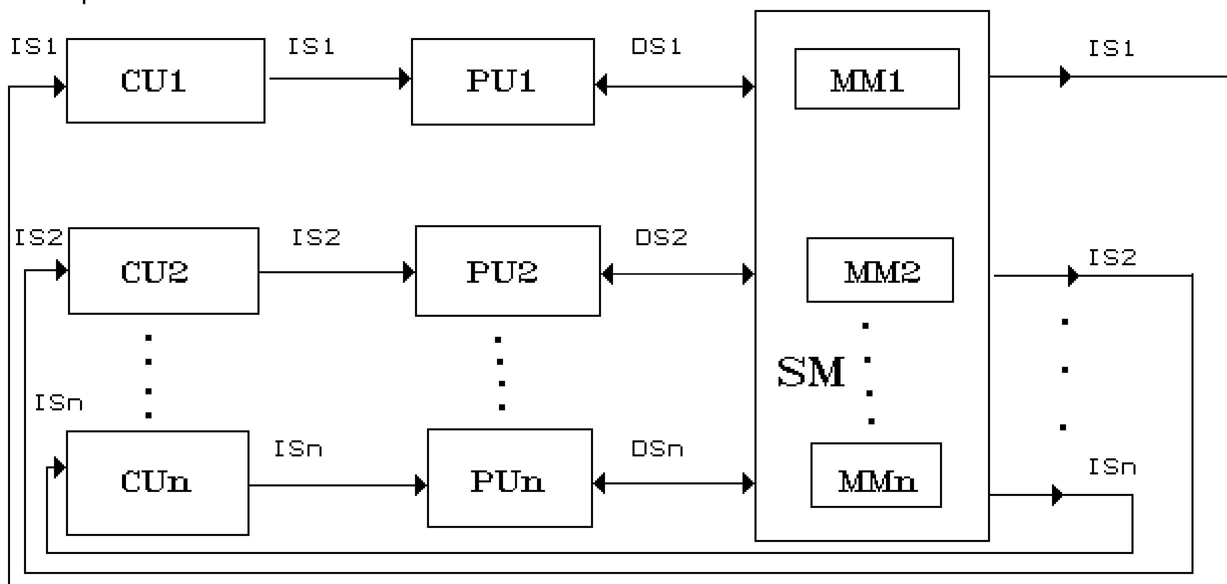


Fig. 4.6. - Computador MIMD.

Varias unidades de control, con varias unidades de ejecución. Es decir, se ejecutan muchas instrucciones con mucho datos, MIMD (Multiple Instruction Multiple Data), siendo éste el caso típico de un sistema de multiprocesamiento.

4.2. - PROCESAMIENTO EN SERIE versus PROCESAMIENTO EN PARALELO.

Tsé-Yun Feng (1972) sugirió la utilización del grado de paralelismo para clasificar las distintas arquitecturas de computadoras. La cantidad máxima de bits que pueden ser procesados dentro de una unidad de tiempo por un computador, es lo que se denomina máximo grado de paralelismo.

Consideremos un conjunto de m palabras de n bits cada una, llamamos un bit-slice a una columna vertical de bits de un conjunto de palabras en la misma posición (ver figura 4.7).

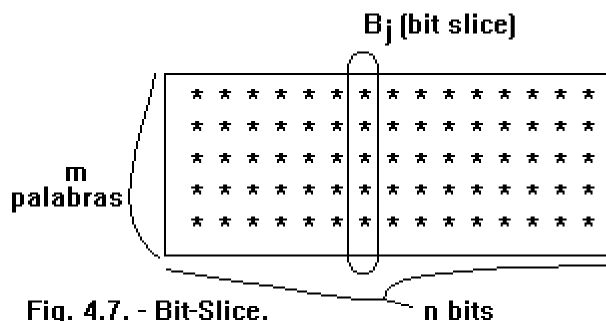


Fig. 4.7. - Bit-Slice.

La figura 4.8 muestra la clasificación de las computadoras por su máximo grado de paralelismo.

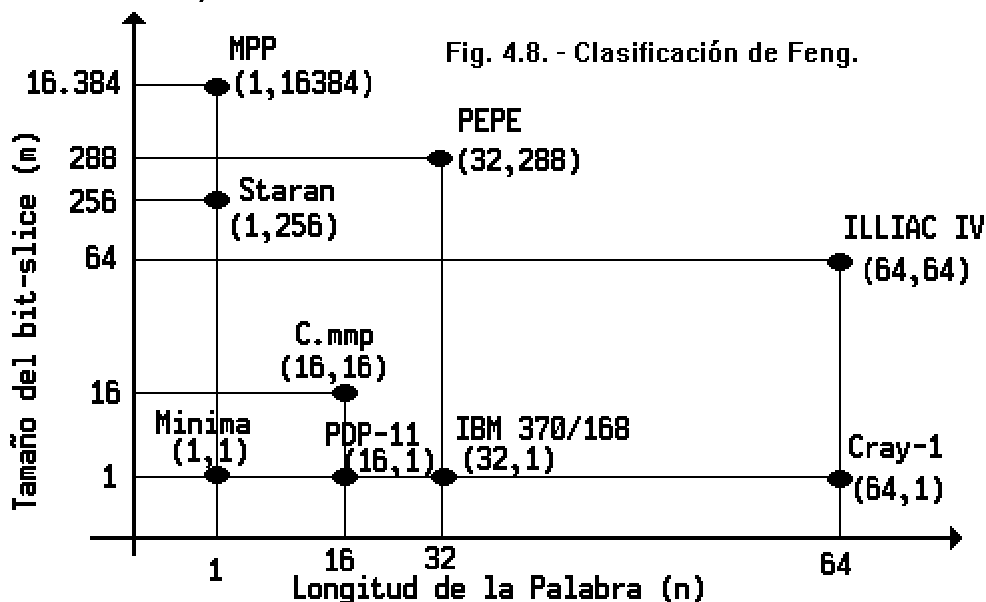
El eje horizontal muestra el tamaño de la palabra n. El eje vertical corresponde al tamaño de la franja de bits m (bit-slice). Ambas medidas están dadas en la cantidad de bits que contiene tanto una palabra o una franja de bits.

El máximo grado de paralelismo P(C) está dado por el producto de m y n:

$$P(C) = m * n$$

De este diagrama se desprende que existen cuatro tipos de procesamiento:

- Palabra en serie y bit en serie (WSBS - Word Serial/Bit Serial)
- Palabra en paralelo y bit en serie (WPBS - Word Parallel/Bit Serial).
- Palabra en serie y bit en paralelo (WSBP - Word Serial/Bit Parallel).
- Palabra en paralelo y bit en paralelo (WPBP - Word Parallel/Bit Parallel).



WSBS ha sido llamado el procesamiento en serie de bits, ya que se procesa un bit por vez (n = m = 1). Es el procesamiento más lento, y solo existió en la primera generación de computadoras.
WPBS (n = 1, m >> 1) ha sido denominado el procesamiento **bis** (de bit-slice) ya que se procesa una porción de m bits a la vez. (Ej.: STARAN).
WSBP (n > 1, m = 1), tal como se encuentra en la mayoría de las computadoras existentes, ha sido llamado **procesamiento de porciones de palabra**, ya que se procesa una palabra de n bits a la vez. (Ej.: arquitectura de Von Neumann).
Finalmente, el WPBP (n > 1, m > 1) es conocido como **procesamiento paralelo total** (o más simplemente como procesamiento en paralelo) en el cual se procesa por vez un arreglo de n x m bits, siendo el más veloz de los cuatro modos vistos.

4.3. - BALANCE DEL ANCHO DE BANDA DEL SUBSISTEMA (BANDWIDTH).

La estructura de bus común tiene el inconveniente que ofrece una comunicación en cuello de botella.
Cada acceso a memoria de un ordenador desaprovecha accesos a millones de bits cuando escoge unos pocos para enviar a través del bus desde la memoria a la unidad central de proceso.
Este despilfarro se tolera por dos razones:
- Primero, porque simplifica nuestro concepto de la máquina y se adapta a nuestra inclinación natural de hacer las cosas una a una.
- Segundo, nos suministra una sola y simple interconexión entre las distintas partes de la máquina.

En general, la CPU es la unidad más veloz dentro de una computadora. Mediremos su ciclo en un tiempo Tp dado en decenas de nanosegundos; el de la memoria, en Tm dado en cientos de nanosegundos; y el del subsistema de E/S, que es el más lento, en Td, dado en unos pocos milisegundos. Entonces:
Td > Tm > Tp
Ej.: En la IBM 370/168, Td = 5 ms (discos); Tm = 320 ns y Tp = 80 ns.
Se define el **ancho de banda o bandwidth** de un subsistema como la cantidad de operaciones realizadas por unidad de tiempo. Para el caso de memoria es la cantidad de palabras de memoria que pueden accederse por unidad de tiempo.
Es importante al hacer mediciones comparativas entre los anchos de banda de los diferentes subsistemas tener bien en claro ciertos conceptos.
Cuando se mide el ancho de banda de la memoria (o equivalentemente el del subsistema de E/S) su visualización no es complicada debido a que es bastante sencillo pensar que, ya que la memoria en un dispositivo pasivo, lo que se mide es la cantidad de información (bytes, palabras, instrucciones, etc.) que la memoria puede transmitir en una cierta unidad de tiempo.
En cambio hablar del ancho de banda de la CPU, que es un dispositivo activo, significa en cierta forma la capacidad de proceso de la misma, es decir más burdamente hablando, que cantidad de instrucciones puede ejecutar por unidad de tiempo.
Si lo que se desea es comparar los anchos de banda de la memoria y la CPU debe ponerse especial cuidado en qué patrón de comparación se está utilizando. Por ejemplo, utilizando los guarismos anteriores, si la IBM 370/168 ejecuta 50 instrucciones en 80 ns su memoria demora 320 ns en transferir esas 50 instrucciones a la CPU.
Sea W la cantidad de palabras que se obtienen por cada ciclo de memoria Tm; luego, el ancho de banda máximo de la memoria es:
Bm = W / Tm (Palabras o bytes)
Ej.: La IBM 3033 tiene un ciclo de procesador de Tp = 57 ns. Por cada ciclo de memoria de Tm = 456 ns pueden obtenerse 8 palabras dobles de 8 bytes c/u. a partir de un sistema de memoria con 8 elementos lógicos de almacenamiento en forma intercalada (interleaved). Luego:
Bm = 8 * 8 bytes / 456 ns = 134 Mb/s
Pero debido a esta partición lógica de la memoria pueden producirse determinados conflictos al querer acceder a una posición, y por lo tanto, el ancho de banda útil (Bmu) será menor:
$Bmu \leqslant Bm$
Se sugiere una medida del tipo:
$Bmu = Bm / M^{1/2}$
Donde M es la cantidad de módulos de memoria del sistema de memoria. Luego, en el ejemplo anterior, se tiene:
$Bmu = 134 / 8^{1/2} = 47.3$ Mb/s
En cuanto a dispositivos externos, el concepto de bandwidth se complica un poco. Piense que, por ejemplo, según los tiempos de latencia y rotación, la tasa de transferencia de una unidad de disco puede variar. En general nos referimos a la tasa de transferencia promedio de una unidad de disco como el bandwidth Bd del disco; un valor típico es del orden de 3 Mb/s. En una unidad de cinta el valor típico ronda los 1.5 Mb/s. Las impresoras, lectoras y terminales son mucho más lentas aún. Usando múltiples drivers estas tasas aumentan.
El bandwidth de un procesador se mide como el máximo porcentaje de cómputo de la CPU; por ejemplo, 160 megaflops (millones de instrucciones de punto flotante por segundo) en los computadores Cray-1, y de 12.5 MIPS (millones de instrucciones por segundo) en los computadores IBM 370/168. Estos son valores pico obtenidos de la división de 1/Tp = 1/12.5 ns y 1/80 ns respectivamente.

En la realidad, el porcentaje útil de CPU es Bpu menor o igual a Bp. Este porcentaje de utilización se calcula sobre la cantidad de resultados (o palabras) por segundo:

$$Bpu = Rw / Tp \quad \text{(palabras/s)}$$

donde Rw es la cantidad de resultados medidos en palabras, y Tp es el tiempo necesario para generar esos resultados Rw.

Por ejemplo, el CDC Cyber-205 tiene un porcentaje pico de 200 MFLOPs para resultados de 32 bits, y solamente de 100 MFLOPs para resultados de 64 bits.

En las computadoras actuales se observa la siguiente relación entre los bandwidths de los distintos subsistemas:

Bm ⩾ Bmu,
Bmu ⩾ Bp,
Bp ⩾ Bpu, y
Bpu ⩾ Bd

Esto significa que la memoria principal tiene el mayor bandwidth, ya que debe ser accedida tanto por la CPU como por los dispositivos de E/S, es decir que del porcentaje efectivo de información transferida por unidad de tiempo por la memoria una parte se dirige a la CPU y otra corresponde al intercambio de información con los medios externos. Por lo tanto necesitamos igualar la potencia de procesamiento de los tres subsistemas. A continuación describimos los dos mecanismos más usados:

4.3.1. - Balance entre CPU y Memoria.

La diferencia de velocidad entre la CPU y la memoria puede achicarse mediante el uso de memoria cache de alta velocidad. La cache tiene un tiempo de acceso Tc = Tp. Un bloque de palabras de memoria es movido a la cache (Por ej. bloques de 16 palabras en la IBM 3033) de tal manera que los datos e instrucciones están disponibles inmediatamente para su uso; también puede servir como un buffer para instrucciones.

4.3.2. - Balance entre dispositivos de E/S y Memoria.

Pueden utilizarse canales de diferentes velocidades entre los dispositivos lentos y la memoria. Estos canales de E/S realizan funciones de bufferización y multiplexamiento para la transferencia de datos desde muchos discos y la memoria principal mediante el robo de ciclos a la CPU. Incluso pueden utilizarse controladores de disco inteligentes para filtrar la información irrelevante de las pistas del disco, lo cual aliviaría la saturación de los canales de E/S.

En el caso ideal desearemos alcanzar un balance total del sistema, en el cual el bandwidth de la memoria coincida con la suma del bandwidth del procesador y de los Dispositivos de E/S, es decir:

$$Bpu + Bd = Bmu$$

donde Bpu = Bp y Bmu = Bm han sido maximizados ambos.

4.4. - Paralelismo con una sola CPU

Procesamiento paralelo es un término utilizado para denotar operaciones simultáneas en la CPU con el fin de aumentar su velocidad de cómputo. En lugar de procesar cada instrucción secuencialmente como en las arquitecturas convencionales, un procesador paralelo realiza tareas de procesamiento de datos e instrucciones concurrentemente.

Para lograr concurrencia en un sistema con un solo procesador se utilizan técnicas de paralelismo que se consiguen multiplicando los componentes de hardware, o técnicas de pipelining.

Introduciremos las características básicas de los **computadores paralelos**, que son aquellos sistemas que enfatizan el procesamiento en paralelo. Estos computadores se pueden dividir en tres configuraciones según la arquitectura:

- Computadores Pipeline (tratados en el presente capítulo).
- Procesadores Matriciales - Array Processors (ver capítulo 5).
- Sistemas Multiprocesadores (ver capítulo 6).

Un computador pipeline hace operaciones superpuestas para explotar el **paralelismo temporal**. Un Array Processor usa ALUs múltiples sincronizadas, para lograr **paralelismo espacial**. Un sistema multiprocesador logra **paralelismo asincrónico** a través de un conjunto de procesadores que interactúan y comparten recursos (periféricos, memorias, bases de datos, etc.).

El pipelining es una forma económica de hacer paralelismo temporal en computadoras. La idea es la misma que la de las líneas de montaje de las plantas industriales. Se divide la tarea en una secuencia de subtareas, cada una de las cuales se ejecuta en una etapa de hardware especializada que trabaja concurrentemente con otra de las etapas del pipeline. Esto permite aumentar el throughput del sistema de forma considerable.

4.5. - COMPUTADORES PIPELINE

Veamos el ejemplo de un pipeline de cuatro etapas: el proceso de ejecución de una instrucción en un computador digital envuelve cuatro pasos principales: levantar la instrucción de memoria (Instruction Fetch - IF); identificar la operación que debe efectuarse (Instruction Decoding - ID); levantar los operandos si son necesarios en la ejecución (Operand Fetch - OF); y ejecutar la operación aritmético lógica que ha sido decodificada. Antes de comenzar a ejecutar una nueva instrucción deben completarse estos cuatro pasos.

La idea de un computador pipeline la vemos en la figura 4.9 : hay cuatro etapas IF, ID, OF y EX ordenadas de forma de una "cascada lineal". Las instrucciones sucesivas se ejecutan de forma superpuesta. La diferencia entre la ejecución superpuesta de instrucciones y la ejecución no superpuesta secuencial se muestra en los diagramas de espacio/tiempo de las Fig. 4.10 y 4.11.

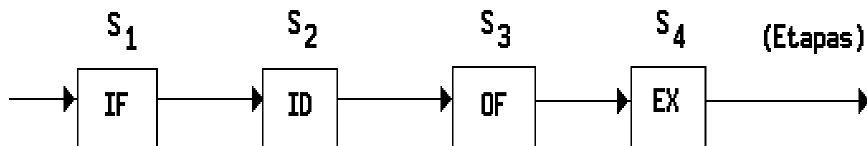


Fig. 4.9. - Un procesador pipeline.

Cada "columna" del gráfico representa un ciclo de pipeline, que es aproximadamente igual al tiempo que tarda la etapa más lenta. Al computador sin pipeline le toma cuatro ciclos de pipeline completar una instrucción, en cambio, un pipeline produce un resultado de salida por cada ciclo luego de cargado el pipe. El ciclo de instrucción ha sido reducido efectivamente a un cuarto del tiempo de ejecución original, por medio de las ejecuciones superpuestas.

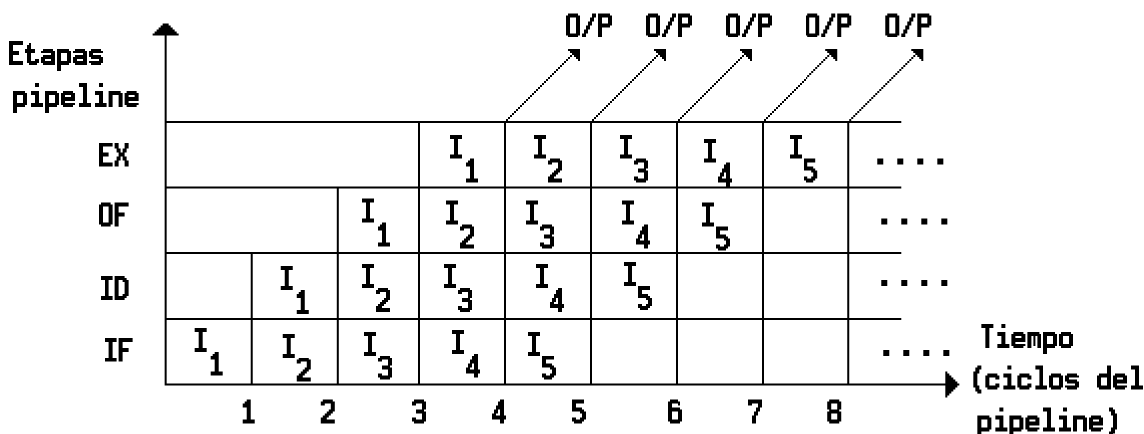


Fig. 4.10. - Diagrama espacio-temporal para un procesador pipeline.

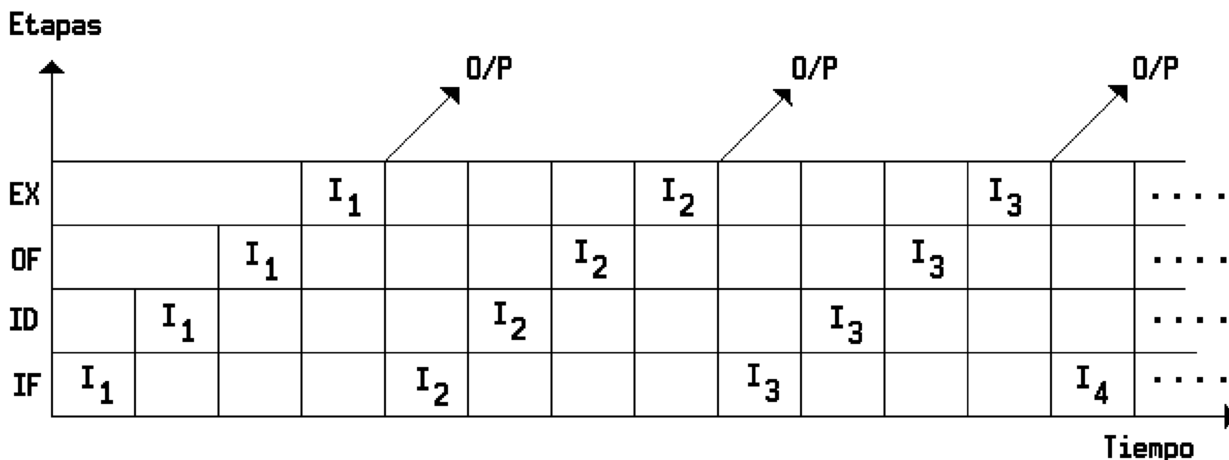


Fig. 4.11. - Diagrama espacio-temporal para un procesador no-pipeline.

Lo que hemos descripto anteriormente, es un **pipeline de instrucción**.

La **máxima velocidad** a la cual pueden ingresar las instrucciones al pipeline depende exclusivamente **del tiempo máximo** requerido para **atravesar una etapa** y **del número de ellas**.

Hay ciertas dificultades que impedirán al pipeline operar con su máxima velocidad. Los segmentos pueden tomar tiempos diferentes para cumplir su función sobre los datos que llegan. Algunos segmentos son salteados por ciertas instrucciones. Por ejemplo, una instrucción modo registro no necesita un cálculo de dirección de operando; por otra parte, dos o más segmentos pueden requerir acceso a memoria al mismo tiempo, haciendo que un segmento tenga que esperar hasta que el otro
termine. (Los conflictos de acceso a memoria se resuelven a menudo utilizando técnicas de interleaving).

4.5.1. - Principios de pipelining lineal

Las líneas de montaje de las plantas han sido utilizadas para aumentar productividad. Su forma original es una línea de flujo (pipeline) de estaciones de montaje en donde los ítems se ensamblan continuamente.

Idealmente todas las etapas tienen que tener igual velocidad de procesamiento, porque sino la más lenta se transforma en un cuello de botella de todo el pipe. La subdivisión de la tarea de entrada en una secuencia apropiada de subtareas es un factor crucial para determinar la performance del pipeline.

En un pipeline de tiempo uniforme, todas las tareas tienen igual tiempo de procesamiento en todas las estaciones. Sin embargo, en la realidad las estaciones sucesivas tienen tardanza de tiempo distinta. La partición óptima del pipeline depende de un número de factores, incluyendo la calidad de las unidades de trabajo (eficiencia y capacidad), la velocidad deseada y la efectividad en costo de toda la línea.

El tiempo que se tarda en obtener resultados continuos es lo que se conoce como **"tiempo de carga del pipe"**.

Esto se traduce en una mejora de la performance, ya que si una función (no sólo instrucciones) se lleva a cabo en T segundos, en la manera convencional, al utilizar un procesador pipeline de N etapas, esa misma función podrá realizarse en T/N segundos.
Obviamente este es un resultado teórico, pues dependerá de la cantidad de operaciones que pueden estructurarse en pipeline y la calidad (pueden ser aritmética de punto flotante, ciclo de instrucciones, etc.).

Dada una tarea T, la podemos subdividir en un conjunto de subtareas {T1,...,Tk}. La relación de precedencia de este conjunto implica que una tarea Tj no puede comenzar hasta que otra tarea anterior Ti ($i < j$) no haya terminado. Las interdependencias de todas las subtareas forman el **grafo de precedencia**. Con una relación de precedencia **lineal**, la tarea Tj no puede comenzar hasta que todas las subtareas anteriores {Ti para todo i no mayor a j} terminen. Un **pipeline lineal** puede procesar una sucesión de subtareas que tengan un grafo de precedencia lineal.

En la figura 4.12 se muestra la estructura básica de un pipeline lineal. El mismo consiste de una cascada de etapas de procesamiento. Las etapas son circuitos que efectúan operaciones aritméticas o lógicas sobre el conjunto de datos que fluyen a través del pipe. Están separadas por registros de muy
alta velocidad que almacenan los resultados intermedios entre etapas, llamados **latches**. La información que fluye entre etapas adyacentes está bajo el control de un reloj común, que se aplica a todos los latches simultáneamente.

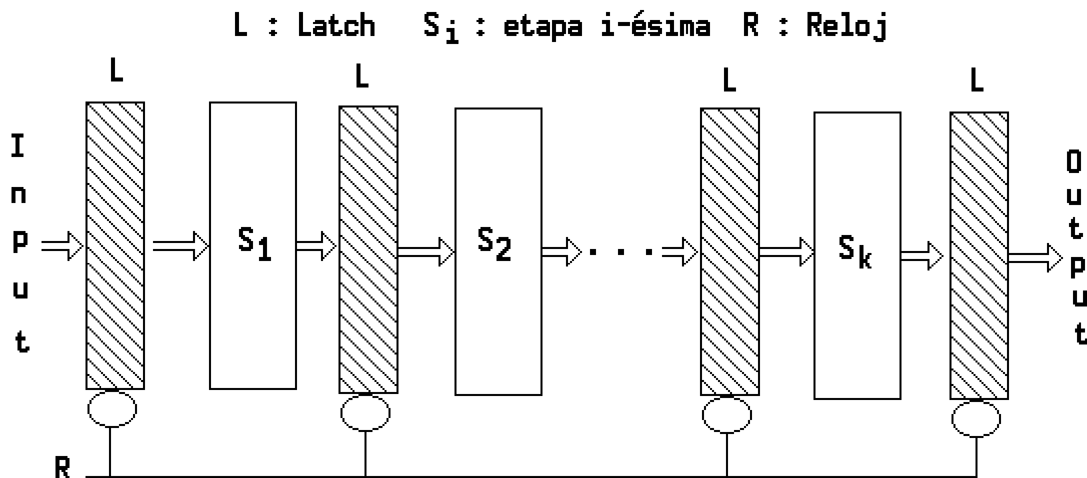


Fig. 4.12. - Estructura de un procesador pipeline lineal.

Para poder aplicar pipeline, o sea la partición en subfunciones de una función se deben dar las siguientes condiciones :

1.- La evaluación de la función es equivalente a la evaluación secuencial de las subfunciones.
2.- La entrada a una subfunción proviene exclusivamente de subfunciones previas en la secuencia de evaluación.
3.- Excepto el intercambio de E/S, no existe otra vinculación entre las subfunciones.
4.- Disponer del hardware suficiente para evaluar las subfunciones.
5.- El tiempo de evaluación de cada subfunción es aproximadamente el mismo.

El flujo de datos dentro del pipeline es **discreto** y se desplazan de etapa en etapa sincronizados por un reloj.

Para evitar que los datos de una etapa ingresen a la siguiente, antes que haya finalizado el proceso anterior se utilizan elementos de memoria en la Entrada y Salida de cada etapa controlados por un reloj asociado (memoria ó **latch**, como se mostró en la Fig. 4.12).

4.5.1.1. - El período de reloj

La circuitería lógica en cada etapa Si tiene una tardanza de tiempo Ti.

Sea T_l la tardanza en llenar cada latch; llamamos entonces período de reloj del pipeline T a:

$T = \text{máximo } \{T_i\}_{1k} + T_l = T_m + T_l$

La **frecuencia** del período del reloj se define como f = 1 / T.

Para ilustrar las operaciones superpuestas en un procesador pipeline lineal, podemos dibujar un diagrama de espacio-tiempo. En la figura 4.13. se muestra el diagrama de espacio-tiempo de un procesador pipeline de cuatro etapas.

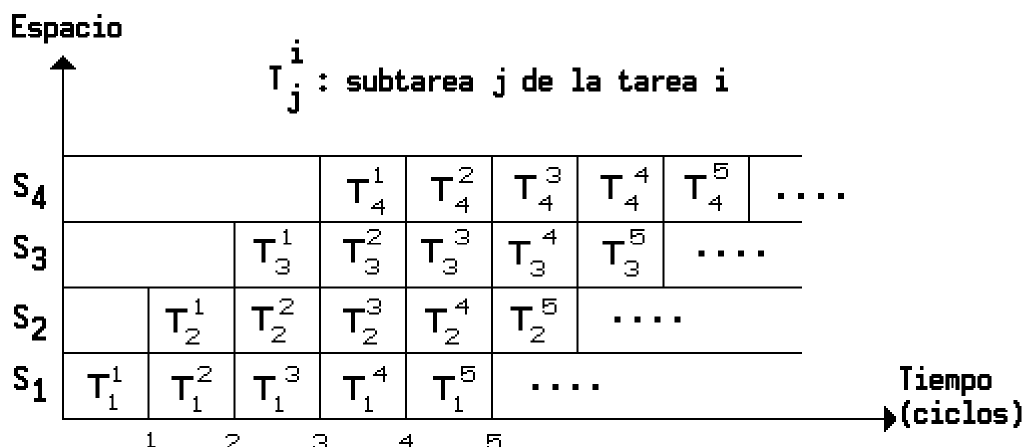


Fig. 4.13. - Diagrama espacio-temporal mostrando el solapamiento de etapas.

Como hemos visto, una vez que el pipe se llena sale un resultado por período de reloj, independientemente del número de etapas en el pipe. Idealmente, un pipeline lineal con k etapas procesa n tareas en $T_k = k + (n-1)$ períodos de reloj. Se usan k ciclos para llenar el pipeline (con la primer instrucción), y hacen falta n-1 ciclos para completar las restantes n-1 tareas. El mismo número de tareas puede ejecutarse en un procesador sin pipeline en un tiempo $T_1 = n * k$.

4.5.1.2. - Aceleración

Definimos la aceleración de un pipeline lineal de k etapas sobre un procesador sin pipeline equivalente como:

$S_k = T_1 / T_k = (n * k) / (k + (n - 1))$

Nótese que la máxima aceleración de Sk tiende a k para n tendiendo a infinito.

En otras palabras, la máxima aceleración de un pipeline lineal es k, donde k es el número de etapas en el pipe, y, en consecuencia, la aceleración es mayor cuantas más instrucciones se puedan procesar.

Esta máxima aceleración nunca se alcanza debido a dependencias de datos entre instrucciones, interrupciones, bifurcaciones del programa y otros factores.

4.5.1.3. - Eficiencia

Si volvemos a la figura 4.13., el producto de un intervalo de tiempo y el espacio de la etapa forman un área llamada el **lapso de espacio-tiempo**. Un lapso de espacio-tiempo determinado puede estar en un estado de Ocupado o Libre, pero no en ambos (Por ej. el lapso de espacio-tiempo 4-S_3 está ocupado por T^3_3).

Utilizamos este concepto para medir la performance de un pipeline: llamaremos la **eficiencia** de un pipeline al porcentaje de lapsos de espacio-tiempo ocupados sobre el total de lapsos de espacio-tiempo (libres más ocupados). Sea n el número de tareas, k el número de etapas del pipeline, y T el período de reloj de un pipeline lineal. Entonces la eficiencia del pipeline se define como:

Eficiencia o EF = (n * k * T) / (k (kT + (n-1) T)
= n / (k + (n - 1))

Notar que si n tiende a infinito entonces la eficiencia tiende a 1. Esto implica que cuanto mayor es el número de tareas que fluyen por el pipeline, mayor es su eficiencia. Además Eficiencia = Sk / k. Esto nos da otra forma de ver la eficiencia: la relación entre la aceleración real y la aceleración ideal k.

4.5.1.4. - Throughput

Es el número de tareas que puede ser completado por un pipeline por unidad de tiempo. Esta tasa refleja el poder de computación de un pipeline. En términos de la Eficiencia y el período de reloj T de un pipeline lineal, definimos el throughput como sigue:

Throughput o TH = n / (k * T + (n-1) * T)
= EF / T

donde n es igual al número total de tareas procesadas durante un período de observación k * T + (n-1) * T.

En el caso ideal, TH = 1/T = f, cuando EF tiende a 1. Esto significa que el máximo throughput de un pipeline lineal es igual a su frecuencia, la que corresponde a **un** resultado de salida por período de reloj.

4.5.1.5. - Pipeline versus Solapamiento.

Si bien tienen significados parecidos y en muchos casos los usaremos como sinónimos no son exactamente equivalentes.
Las condiciones del **Pipeline** son :

1.- Cada evaluación de funciones básicas es independiente de la anterior.
2.- Cada evaluación requiere aproximadamente la misma secuencia de subfunciones.
3.- Cada subfunción se encadena perfectamente con la anterior.
4.- Los tiempos de las subfunciones son aproximadamente iguales.

Las condiciones de **Solapamiento** son :

1.- Existe dependencia entre las distintas evaluaciones (funciones).
2.- Cada evaluación puede requerir una secuencia diferente de subfunciones.
3.- Cada subfunción tiene un propósito distinto.
4.- El tiempo de cada evaluación no es necesariamente constante, sino que depende de la función y de los datos que la atraviesan.

El primer caso se lo suele denominar **Pipeline Sincrónico o Unifunción** y el segundo **Pipeline Asincrónico o Multifunción.**

Un ejemplo de Pipeline Unifunción está dado por una unidad destinada a realizar sumas en punto flotante. El resultado de una suma **no** depende de la anterior y la secuencia de etapas de las sumas es siempre la misma.

Un ejemplo de Pipeline Multifunción es la ejecución de instrucciones en un procesador, donde cada instrucción puede desarrollar caminos diferentes a través del Pipeline, por ejemplo al decodificarse una instrucción de salto la próxima etapa que debe continuar para dicha instrucción no consiste en obtener la o las direcciones de los operandos sino en alterar la dirección de la próxima instrucción que debe ingresar en el pipe, vaciando (flush) previamente aquellas instrucciones que ingresaron hasta el momento en que se detectó la instrucción de salto.

4.6. - CLASIFICACIONES DE PROCESADORES PIPELINE.

De acuerdo a los niveles de procesamiento, Händler (1977) ha propuesto el siguiente esquema de clasificación para los procesadores pipeline:

4.6.1. - **Pipelines aritméticos**

La ALU de un computador puede dividirse para hacer operaciones de pipeline en varios formatos. Hay ejemplos bien claros en los pipelines usados en la Star-100, TI-ASC, Cray-1 y Cyber 205 (Fig. 4.14).

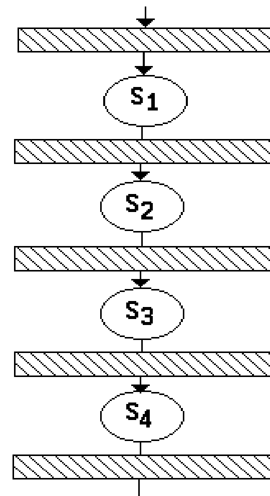


Fig. 4.14. - Pipeline Aritmético.

4.6.2. - **Pipelines de instrucción**

La ejecución de un flujo de instrucciones puede hacerse en forma de pipeline, como vimos en la primer parte del capítulo, superponiendo la ejecución de la instrucción actual con las acciones de levantar, decodificar instrucciones y levantar operandos.

Esta técnica también es conocida con el nombre de **lookahead de instrucciones**. Casi todos los computadores de alta performance actuales tienen pipelines de instrucción (Ver Fig. 4.15).

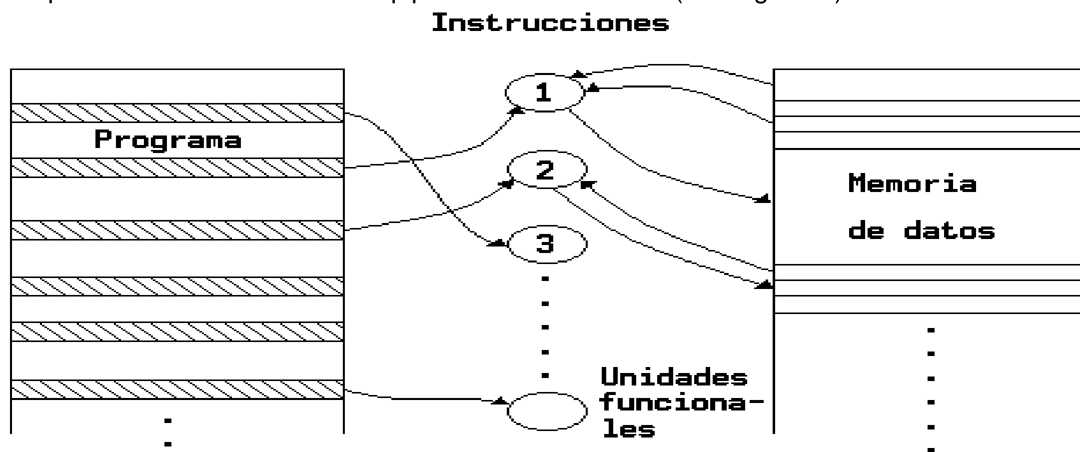


Fig. 4.15. - Pipeline de instrucciones.

4.6.3. - **Pipelines de procesador**

Se refiere al procesamiento del mismo flujo de datos por una cascada de procesadores, cada uno de los cuales procesa una tarea específica. El flujo de datos pasa el primer procesador, cuyo resultado se almacena en un bloque de memoria intermedia, que también es accesible por el segundo procesador. Este pasa el resultado refinado al tercero, y así siguiendo. En el momento en que Händler escribió su clasificación de los pipelines aún no existían implementaciones de pipelines de procesador (Fig. 4.16).

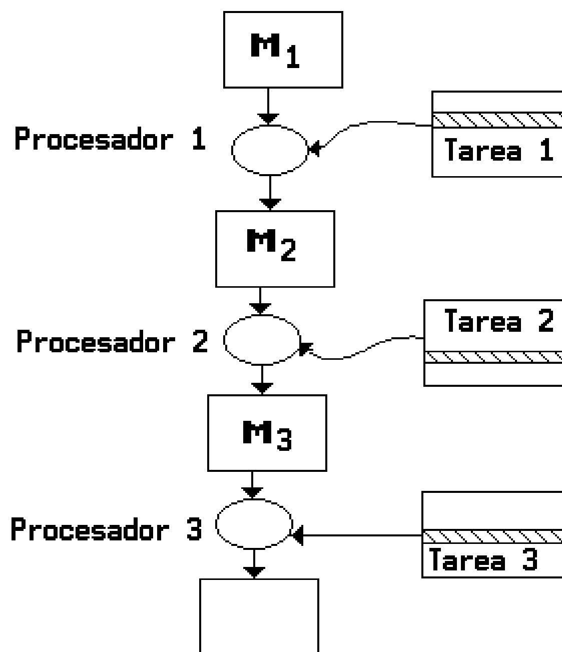


Fig. 4.16. - Pipeline de procesadores.

De acuerdo a las configuraciones del pipeline y estrategias de control, Ramamoorthy y Li (1977) han propuesto los siguientes tres esquemas de clasificación:

4.6.4. - **Pipelines unifuncionales versus multifuncionales**

Un pipeline unifuncional es aquel que tiene una sola función y dedicada, como el que mostraremos en el Ejemplo 1 (sumador de punto flotante).

Un pipeline multifuncional puede efectuar distintas funciones, por medio de interconexiones de varios subconjuntos de etapas en el pipeline, ya sea en diferentes momentos o al mismo tiempo, aquí además de datos existen señales de control que indican al pipeline las funciones a cumplir. El camino a seguir está determinado por las mismas instrucciones (IBM 360/91).

Esto es equivalente a un cambio de **configuración**.

4.6.5. - **Pipelines estáticos vs. dinámicos**

Un pipeline estático puede tener una sola configuración funcional por vez. Los pipelines estáticos pueden ser unifuncionales o multifuncionales.

El pipelining es posible en los pipelines estáticos sólo si existen instrucciones del mismo tipo que se ejecutan una seguida de la otra. La función de un pipeline no debería cambiar frecuentemente, sino su performance sería muy baja.

Un procesador pipeline dinámico permite distintas configuraciones funcionales existiendo simultáneamente. Por lo tanto, un pipeline dinámico tiene que ser multifuncional. Y por otro lado, un pipeline unifuncional tiene que ser estático. La configuración dinámica requiere un mayor control que los estáticos y además mecanismos de secuenciamiento.

4.6.6. - **Pipelines escalares vs. vectoriales**

Dependiendo de los tipos de instrucción o datos, los procesadores pipeline pueden clasificarse en escalares o vectoriales. Un pipeline escalar procesa una secuencia de operandos escalares bajo el control de un ciclo (por ejemplo un DO de Fortran). Generalmente se hace prefetching de las instrucciones de un loop pequeño y se almacena en el buffer de instrucción.

Los operandos necesarios para instrucciones escalares repetidas se almacenan en una cache de datos para siempre tener el pipeline cargado con operandos. El IBM 360/91 es una máquina equipada con pipelines escalares (aunque no tiene una cache).

Los pipelines vectoriales están diseñados especialmente para manejar instrucciones vectoriales sobre operandos vectoriales. Los computadores con instrucciones vectoriales también suelen llamarse **procesadores vectoriales**. El manejo de operandos vectoriales está hecho por medio de firmware o hardware, en lugar del control de software que tienen los pipelines escalares. La ventaja fundamental es que se levanta y se decodifica la instrucción una sola vez por cada par de vectores, lo que ahorra gran cantidad de tiempo. Existen diversos pipelines vectoriales: el TI-ASC, STAR-100, Cyber-205, Cray-1, etc. Para mayor detalle remitirse al punto 4.11 del presente capítulo.

4.7. - PIPELINES GENERALES Y TABLAS DE RESERVACIÓN

El uso eficiente de un pipeline requiere :
- Flujo de datos permanente a través del mismo
- Controlar y organizar el flujo de datos evitando conflictos internos en las etapas del pipeline.

Si fuesen todos lineales **no** habría problemas y el ruteo sería trivial. Pero las etapas no son todas iguales en duración y además pueden ser reconfigurados los Pipes, luego es necesario establecer procedimientos para ruteo y control.

A pesar de estas dificultades pueden hallarse algoritmos de ruteo que requieren del siguiente entorno :

1.- El tiempo de ejecución de todas las etapas es un múltiplo de un reloj.

2.- Una vez que el proceso comenzó en el pipeline el diagrama temporal de utilización de las etapas por los distintos datos de entrada queda fijo.

Lo que hemos estudiado hasta ahora son pipelines lineales, sin conexiones feedback. Las entradas y salidas de estos pipelines son totalmente independientes. En algunos tipos de cálculos, las salidas de un pipeline se utilizan como futuras entradas. En otras palabras, las entradas pueden depender de salidas anteriores. En estos casos la historia de utilización del pipeline determina el estado actual del mismo. Estos pipelines pueden tener un flujo de datos no lineal.

Como ejemplo, en la figura 4.17 mostramos un pipeline que tiene conexiones feedback y feedforward.

Llamaremos S1, S2 y S3 a las etapas. Las conexiones entre etapas adyacentes forman la cascada original del pipeline. Una conexión feedforward conecta dos etapas Si y Sj con j mayor o igual a i+2; y una conexión feedback es aquella que conecta una etapa Si con Sj si j es menor o igual a i.

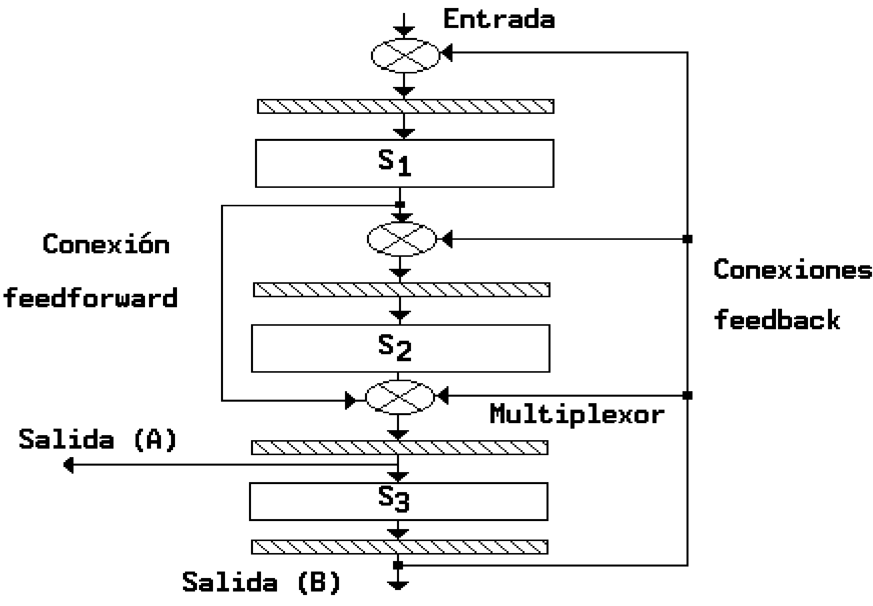


Fig. 4.17. - Un pipeline de ejemplo.

En este sentido, un pipeline lineal "puro" es un pipeline lineal sin conexiones feedback o feedforward.

La coordinación de las entradas feedback y forward es crucial ya que el uso indebido de tales conexiones puede destruir las ventajas inherentes del pipeline. En la práctica, muchos pipelines aritméticos permiten conexiones no lineales como un mecanismo para implementar recursividad y funciones múltiples.

Llamaremos a estos pipelines con conexiones feedforward o feedback, **pipelines generales**. Utilizaremos un gráfico bidimensional para mostrar cómo se utilizan las etapas sucesivas del pipeline en ciclos sucesivos. Llamaremos a este gráfico una **tabla de reserva o tabla de reservación**.

Las dos tablas de reserva que se muestran en la figura 4.18 corresponden a las dos funciones del pipeline del ejemplo 4.19.

Las filas corresponden a las etapas del pipeline, y las columnas a las unidades de tiempo de reloj. El total de unidades de reloj en la tabla se llama el **tiempo de evaluación** para la función dada. La tabla de reserva representa el flujo de datos a través del pipeline para la evaluación de una función dada.

Una entrada marcada en el cuadro (i,j) indica que la etapa Si se utilizará j unidades de tiempo luego de la iniciación de la evaluación de la función.

El flujo de datos en un pipeline estático y unifuncional puede ser descripto de forma completa con una tabla de reservación. Un pipeline multifuncional puede usar distintas tablas de reserva para funciones distintas. Por otro lado, una tabla de reserva dada no corresponde unívocamente a un pipeline particular. Puede haber distintos pipelines con distintas estructuras de interconexión utilizando la misma tabla de reserva.

Tiempo

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
S_1	A			A			A	
S_2		A						A
S_3			A		A	A		

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
S_1	B				B		
S_2			B			B	
S_3		B		B			B

Fig. 4.18. - Tablas de reservación para dos funciones.

Para visualizar el flujo de datos a través de las vías de datos en un pipeline para su evaluación completa, en la figura 4.19. se muestran los ocho pasos necesarios para evaluar la función A en el pipeline del ejemplo.

Esta función está determinada por la tabla de reservación para la función A dibujada precedentemente (Fig. 4.18).

Como explicamos anteriormente las tablas de reservación son válidas para la evaluación de una función particular. Además una tabla de reservación **no** corresponde a un pipeline determinado, sino que a más de uno.

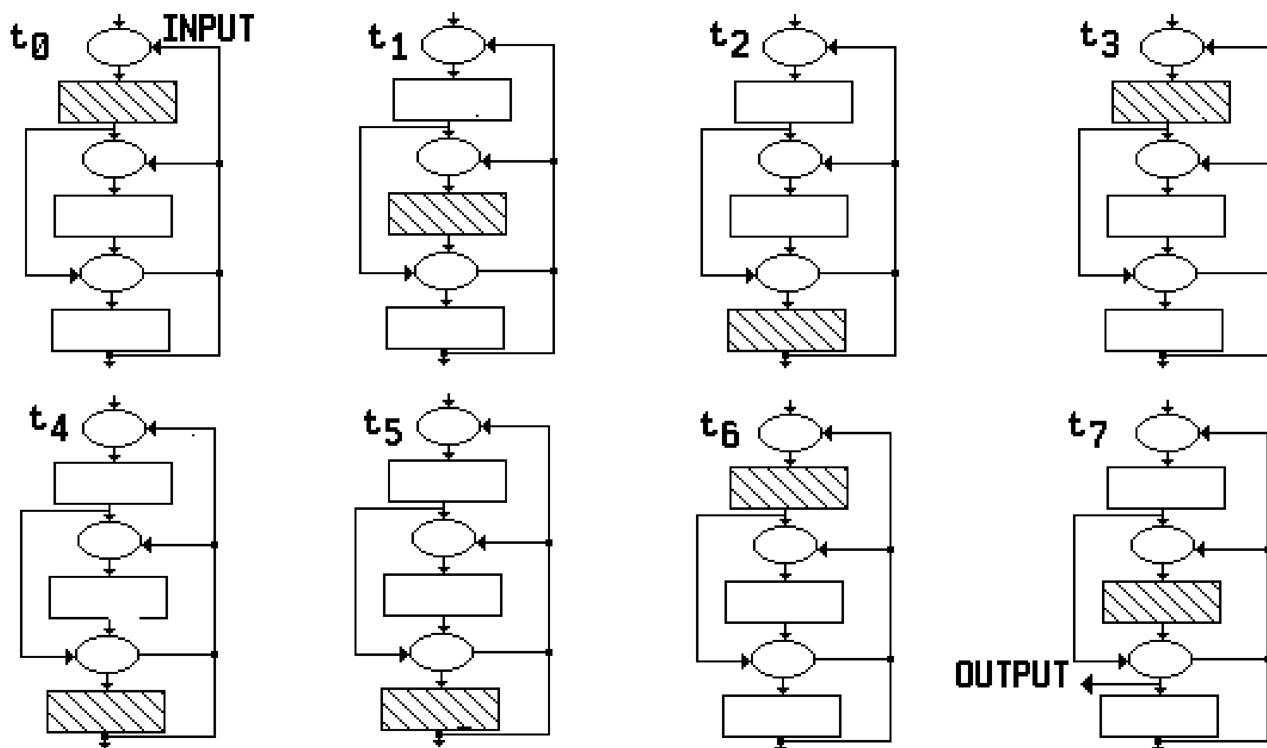


Fig. 4.19. - Ocho pasos utilizando el pipeline de ejemplo para evaluar la función A.

La idea de las Tablas de Reservación es evitar que dos o más datos intenten ingresar en forma simultánea a una misma etapa, lo que provocaría una **colisión**.

Supongamos que tenemos la tabla de reservación de la Fig. 4.20.

Para esta tabla de reservación se podrían tener los diseños de los pipelines de la Fig. 4.21.

En nuestro ejemplo, si se inician procesos en los instantes 0, 2 ó 6 se producirá una colisión.

	Tiempo	0	1	2	3	4	5	6	7
Etapa	1			A		A			
	2		A		A		A		
	3	A		A				A	

Fig. 4.20.

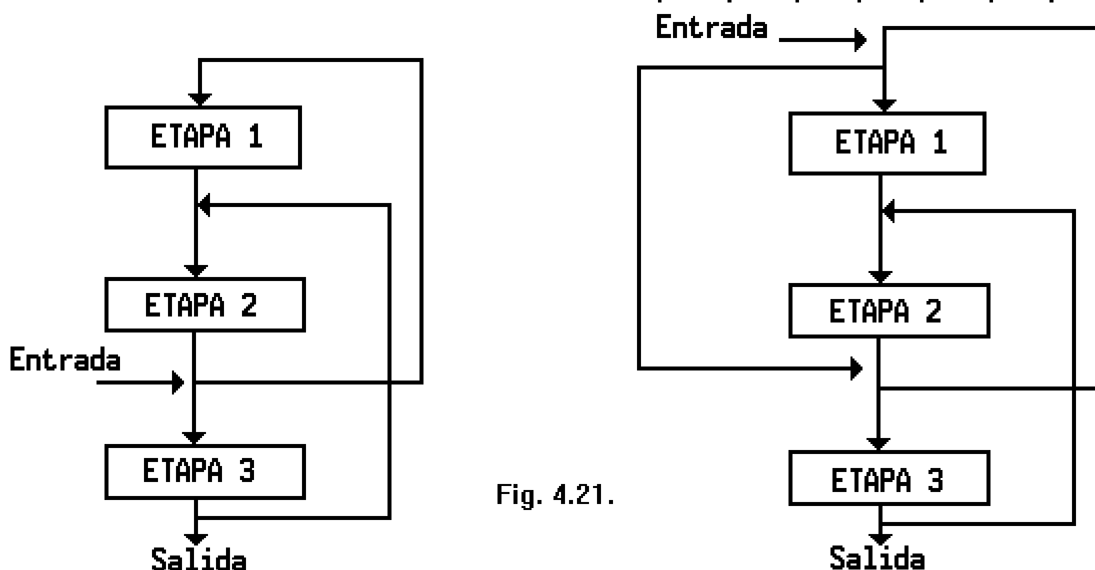


Fig. 4.21.

4.8. **Performance de pipelines.**

Un parámetro clave para la evaluación de la performance de un Pipeline es la "Latencia", que se define como el número de unidades de tiempo que separa dos inicios de la misma o distintas tablas de reservación.

En el caso de los Pipeline unifuncionales la latencia nunca podrá ser cero, ya que todos comienzan por la misma etapa (el menor posible es 1).

En el caso de los Pipeline dinámicos la latencia puede llegar a ser cero, ya que dos tablas de reservación distintas pueden superponerse.

Para lograr una mejor performance en un Pipeline es deseable obtener una "Latencia" promedio mínima.

Hay que tener en cuenta que elegir la mínima latencia entre dos inicios (estrategia **"greedy"**) no siempre llevará a la mejor performance.

Tomemos el ejemplo de la Fig. 4.22.

Nótese que eligiendo la mínima latencia:

- B2 puede comenzar en el intervalo 3 (latencia 3)
- y B3 puede iniciarse en el intervalo 11 (con una latencia 8 respecto del inicio de B2).

Tiempo		0	1	2	3	4	5	6	7
Etapa	1	B	B					B	B
	2			B		B			
	3				B		B		

Fig. 4.22.

Tomando la estrategia "greedy" veremos que nos da una Latencia alternada de 3 y 8 esto implica que :
(3 + 8) / 2 = 5,5 (latencia media 5,5) (Fig.4.23).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	B1	B1		B2	B2		B1	B1		B2	B2	B3	B3		B4	B4		B3	B3		B4	B4
2			B1		B1	B2		B2						B3		B3	B4		B4			
3				B1		B1	B2		B2						B3		B3	B4		B4		

Fig. 4.23. Repetición del ciclo

Ahora observemos una estrategia que comience con "Latencia " 4, esto nos permite un nuevo inicio cada cuatro intervalos :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	B1	B1			B2	B2	B1	B1	B3	B3	B2	B2	B4	B4	B3	B3	B5	B5	
2			B1		B1		B2		B2		B3		B3		B4		B4		
3				B1		B1		B2		B2		B3		B3		B4		B4	

Fig. 4.24. Repetición del ciclo

con lo cual llegamos a una mejor performance.

Visto el problema anterior nos queda cómo establecer una "latencia" óptima.

Para ello se construye un *Vector de Colisiones* (Colisión = 1) que se obtiene estudiando la Tabla de reserva de una determinada tarea.

Se intenta empezar la función en cada una de las columnas del intervalo. Si dentro del intervalo existe colisión se asigna 1 en el vector, caso contrario vale 0. Para nuestro ejemplo (Fig. 4.22) ese vector de colisiones resulta : < 1 1 1 0 0 1 1 1 >.

Ahora se debe simular el paso del tiempo sobre este vector de colisiones (haciendo shifts a izquierda) y comprobar que una configuración similar puede ser incluida en el pipeline sin que ocurran colisiones entre ellas. Para ello se construye un grafo, donde cada nodo es un tick de reloj (shift a izquierda) y se trata de comprobar cuando es posible incluir una tabla de reserva similar por medio de una operación OR con la configuración inicial del vector.

Con los OR se encuentran todos los vectores de colisiones resultantes, el algoritmo, en nuestro ejemplo, no hace más de 7 shifts.

Obviamente estas operaciones OR se darán solamente cuando el vector que se está corriendo comience con un cero (a izquierda); pues de otra manera, independientemente de lo que diga el resultado de la operación OR, habrá colisión.

Los vectores de colisiones derivados se utilizan para prevenir las colisiones en el cálculo de las funciones futuras respecto de las iniciadas previamente, en tanto que el vector de colisiones construido al principio sirve para prevenir colisiones en el cálculo de la función actual.

Contando los niveles de este grafo se obtiene cual es la latencia "mínima óptima promedio" (MAL - Minimum Average Latency).

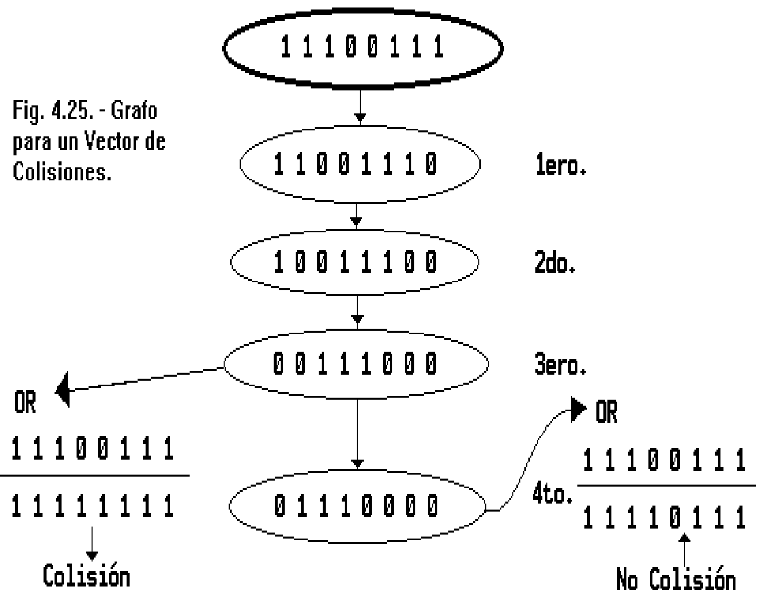


Fig. 4.25. - Grafo para un Vector de Colisiones.

$A = a * 2^p$ $B = b * 2^q$

p a b q

Etapas

S_1 Substractor de exponente — Otra fracción — Selector de fracción — Fracción min(p,q) — Shift a derecha — r=max(p,q) — t=|p-q| — L

S_2 r — Sumador de fracciones — c — L

S_3 Contador para supresión de ceros — c — Shift a izquierda — L — d

S_4 Sumador de exponente — (fracción normalizada) — L

s — $C = d * 2^s = A + B$ — d

Fig. 4.26. - Un sumador pipeline de punto flotante de cuatro etapas de procesamiento.

Veamos una aplicación del método en nuestro ejemplo.
Como estamos en el 4to. nivel del grafo esto implica que la latencia media mínima la obtendremos introduciendo una nueva instrucción con latencia 4.

4.9. - Ejemplificación De Los Principios Operacionales De Los Pipelines.

4.9.1. - **Ejemplo 1:** Ilustraremos el diseño de un pipeline sumador de punto flotante, pipeline unifuncional. (Figura 4.26.)

Este pipeline puede construirse linealmente con cuatro etapas funcionales. Las entradas al pipeline son dos números normalizados en punto flotante:

$A = a * 2^p$ $B = b * 2^q$

Queremos calcular la suma:

$C = A + B = c * 2^r = d * 2^s$;

donde r = máximo(p,q), y d es c normalizado.

Las operaciones que se hacen en las cuatro etapas de pipeline son las siguientes:

1. Comparar los dos exponentes p y q para hallar el mayor exponente r = máximo (p,q) y determinar su diferencia t = p - q en módulo.
2. Hacer shift a la derecha de la fracción del menor exponente t bits para igualar los dos exponentes antes de la suma de fracciones.
3. Sumar ambas fracciones para producir la fracción intermedia c.
4. Contar el número de ceros a izquierda, digamos u, en la fracción c, y hacer shift u bits a la izquierda para producir la fracción normalizada d. Actualizar el exponente mayor s, haciendo s = r - u para producir el exponente de salida.

4.9.2. - **Ejemplo 2:**

La CPU de una computadora digital moderna puede dividirse generalmente en tres secciones, como vemos en la figura 4.27:

- la Unidad de Instrucciones,
- la Cola de Instrucciones y
- la Unidad de Ejecución.

Desde el punto de vista operacional, las tres unidades forman un pipeline. Los programas y datos residen en la memoria principal, que generalmente consiste de módulos de memoria interleaved. La memoria cache guarda copias de datos y programa listos para ejecución, y así acortamos la diferencia de velocidades entre Memoria y CPU.

La Unidad de Instrucciones consiste de etapas de pipeline que hacen

IF, levantar instrucciones de la memoria,
ID, decodificar los códigos de operandos de las instrucciones,
Cálculo de direcciones de operandos, y
OF , levantar los operandos desde la memoria, (si es necesario).

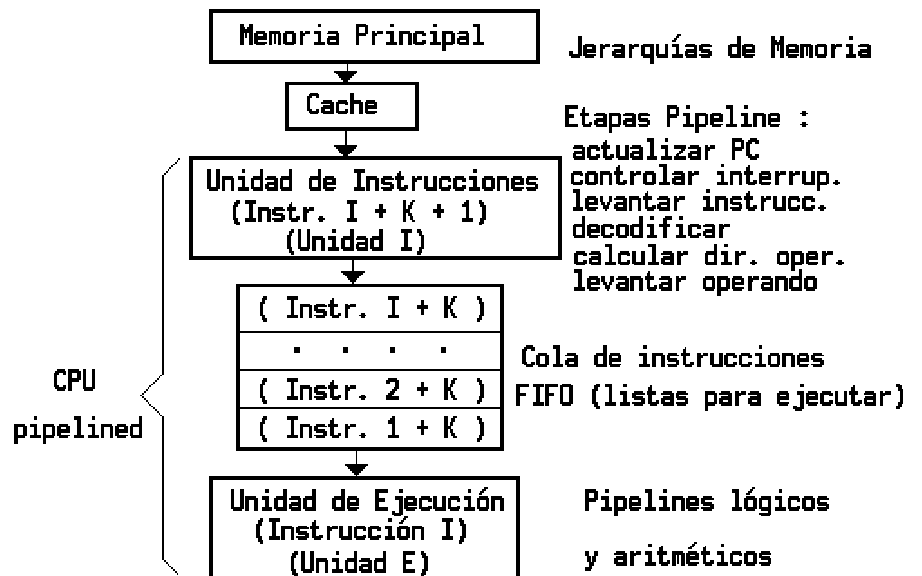


Fig. 4.27. - Estructura típica de una CPU pipelined.

La Cola de Instrucciones almacena instrucciones decodificadas y operandos que han sido levantados de memoria. La Unidad de Ejecución puede contener pipelines con varias funciones que hagan operaciones aritmético-lógicas. Mientras la Unidad de Instrucciones está levantando la instrucción I+K+1, la cola de instrucciones contiene las instrucciones I+1,...,I+K, y la unidad de ejecución ejecuta la instrucción I.

En este sentido, la CPU es un buen ejemplo de un pipeline lineal.

4.10. - **ALGUNOS PROBLEMAS DE LOS PIPELINES.**

En una máquina secuencial tradicional una instrucción **no** comienza hasta que se haya completado la anterior.

Si aplicamos Pipeline a un procesador de este tipo pueden aparecernos 3 clases de riesgos :

Read-after-Write (RAW)

Write-after-Read (WAR)
Write-after-Write (WAW)
Veamos ejemplos en el siguiente código de programa :

```
.....
.....
1) ALMACENAR  X
2) SUMAR   X
.....
3) ALMACENAR  X
.....
.....
4) ALMACENAR  X
.....
```

Riesgo RAW - Si 2) extrae después que 3) escribe
Riesgo WAR - Si 1) escribe después que 2) extraiga
Riesgo WAW - Si 3) escribe después que 4)

Luego si dadas las instrucciones i y j (j posterior a i), y definimos L(i) como el conjunto de todos los objetos leídos por i, y M(i) como el conjunto de todos los objetos modificados por i, resulta que existen condiciones de riesgo cuando se de alguna de las siguientes 3 condiciones :

$M(i) \cap L(j) \neq \phi$ (WAR)

$L(i) \cap M(j) \neq \phi$ (RAW)

$M(i) \cap M(j) \neq \phi$ (WAW)

Para detectar riesgos generalmente, se utiliza la etapa de búsqueda, y se comparan los conjuntos de objetos leídos y modificados por las instrucciones dentro del Pipe.

Para resolver situaciones de riesgo, generalmente, se detiene el Pipe hasta que se ejecute completamente la instrucción que provocaría el riesgo y luego continuar normalmente con las siguientes instrucciones.

Como veremos posteriormente estos riesgos también existen al dividir un programa en varias tareas concurrentes.

4.10.1. - **Prebúsqueda de Instrucciones**

En los ejemplos que hemos visto, hemos considerado que la carga de instrucciones se realiza mientras se está ejecutando otra.

De todas maneras esto es un motivo de riesgo, especialmente en las instrucciones de salto que debería tratarse de la manera que se puede visualizar en la Fig. 4.28.

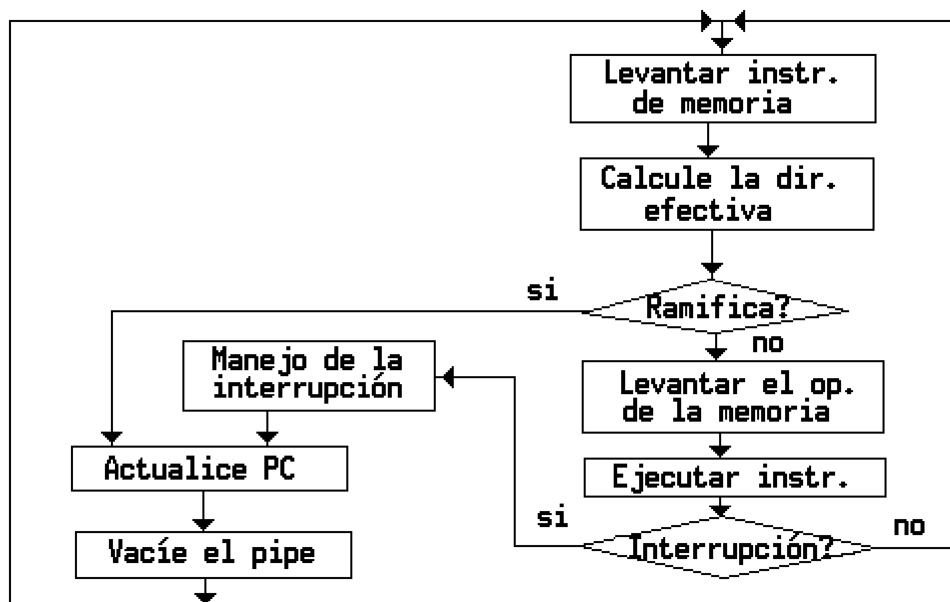


Fig. 4.28. - Efecto de las bifurcaciones en un pipeline de cuatro etapas.

Una forma de graficar esta situación sería considerar al procesador de la forma que se ve en la Fig. 4.29.

Donde la Unidad I realiza la decodificación y cálculo de direcciones y la Unidad E realiza la ejecución de la instrucción.

Si así lo dejamos prácticamente todas las instrucciones serían ejecutadas en la Unidad I y muy pocas en la Unidad E, por ejemplo las sumas.

Veamos de que manera es posible dividir (particionar) la Unidad I, en especial en el manejo de saltos, inclusive adjudicándole más funciones.

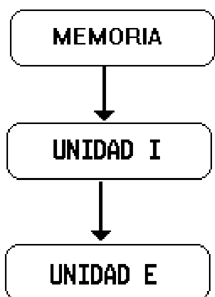


Fig. 4.29.

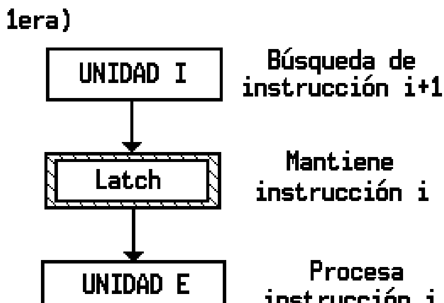


Fig. 4.30.

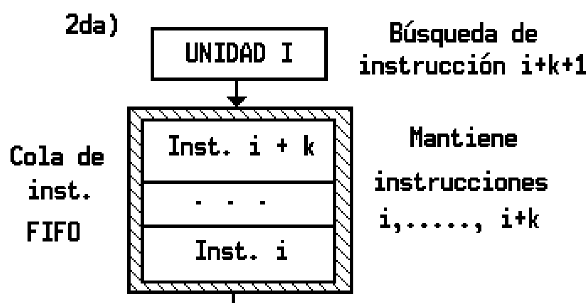


Fig. 4.31.

4.10.2. - **Manejo de saltos**

Las unidades I y E pueden estar básicamente conectadas de 2 formas. La primera se denomina **conexión rígida** (Fig. 4.30). Nótese que la segunda forma de conexión (Fig. 4.31) es mucho más flexible atendiendo que permite un mejor manejo por almacenar varias instrucciones simultáneamente.

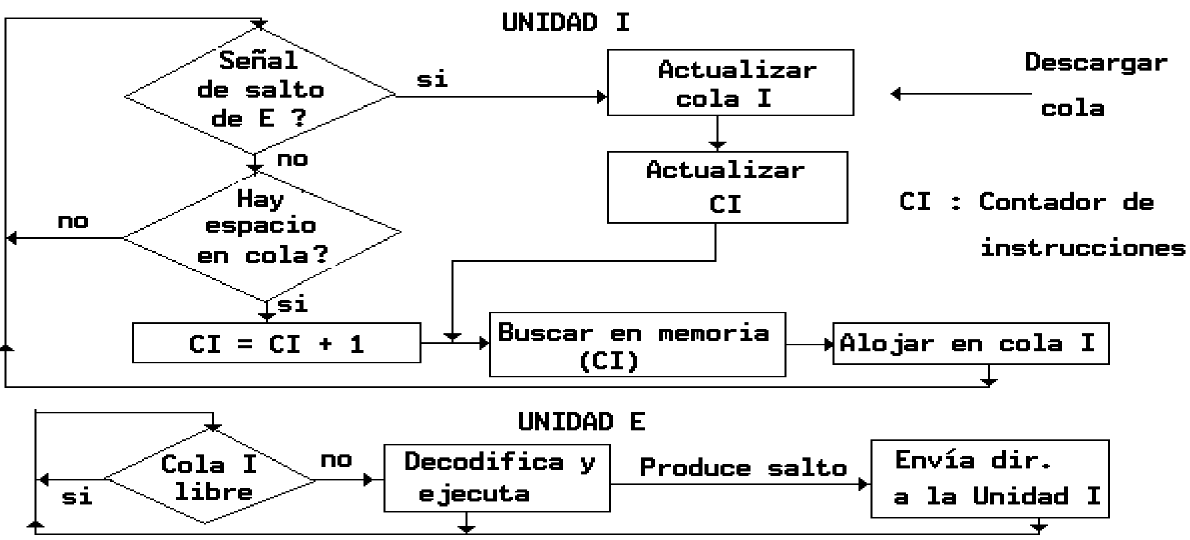


Fig. 4.32.

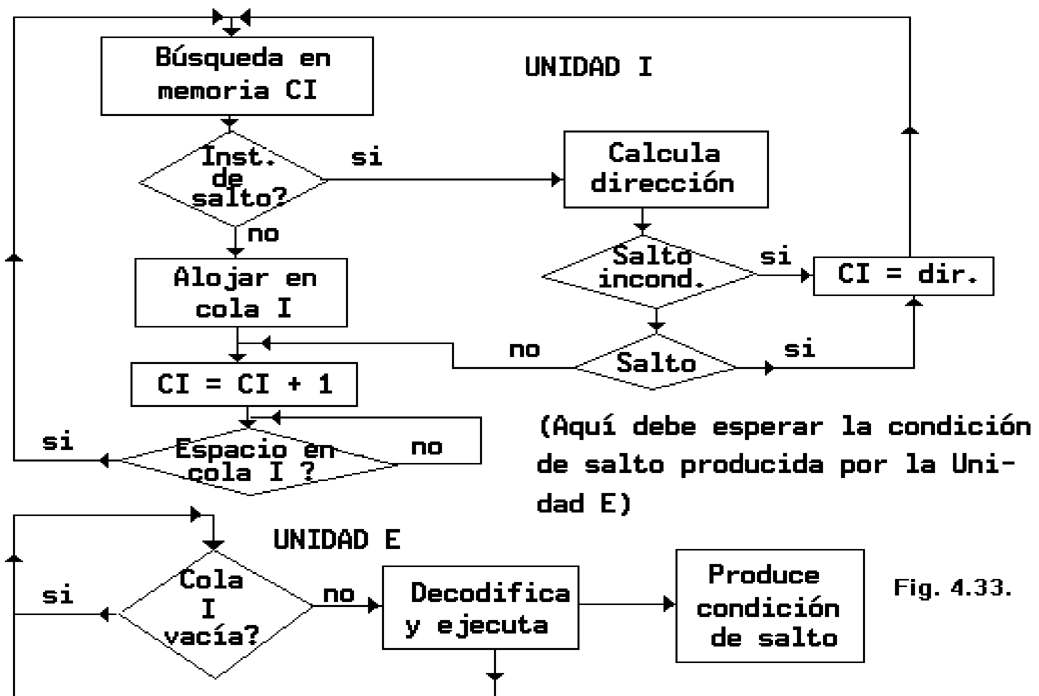


Fig. 4.33.

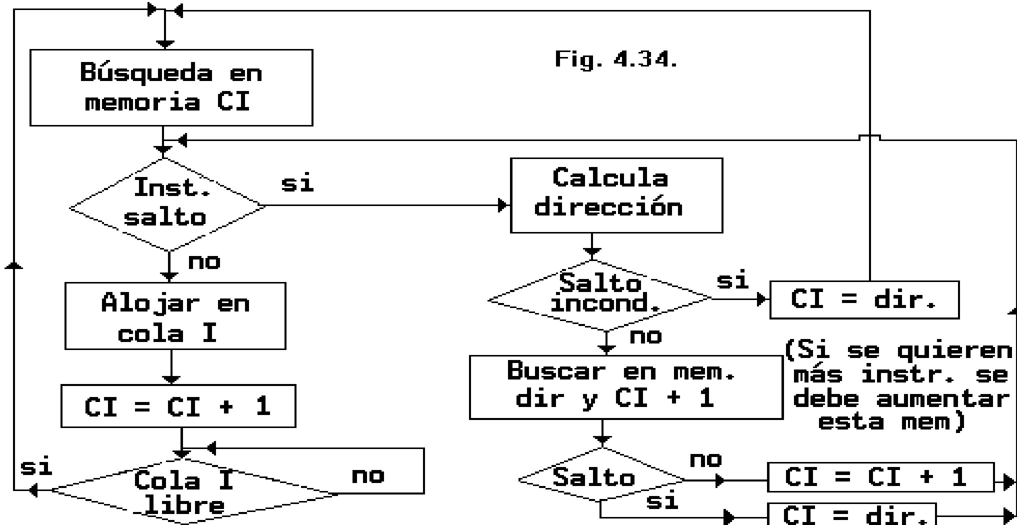
Veamos un diagrama lógico (Fig. 4.32) de cómo actúan ambas unidades, suponiendo que la decodificación la realiza la unidad E, utilizando cola FIFO de instrucciones.

Veamos ahora otro diagrama lógico (Fig. 4.33) de estas unidades permitiendo que la Unidad I decodifique en parte las instrucciones, para detectar saltos y lógicamente debe generar direcciones.

En el caso anterior el "descargar cola" no es necesario, luego queda resolver el problema de la espera de la Unidad I mientras se espera la condición producida por la Unidad E.

Una solución es realizar la prebúsqueda en ambas ramas posibles y descartar luego la que no cumple la condición.

Un diagrama posible de la **Unidad I** para este caso sería el que puede visualizarse en la Fig. 4.34.



Un ejemplo : el caso del Intel 8086 es un pipeline de 2 etapas, una de prebúsqueda de instrucciones y otra de ejecución, interconectadas por una cola de instrucciones.

Para determinar el tamaño de la cola de instrucciones se realizaron simulaciones que determinaron que el tamaño mayor de 6 bytes **no** producía mejoras, luego se eligió esta cifra que da un valor promedio de 2 instrucciones.

Hay que tener en cuenta que para que un procesador pipeline sea eficiente es necesario que pueda acceder a memoria toda vez que sea necesario, o sea que se necesita un apropiado ancho de banda de memoria.

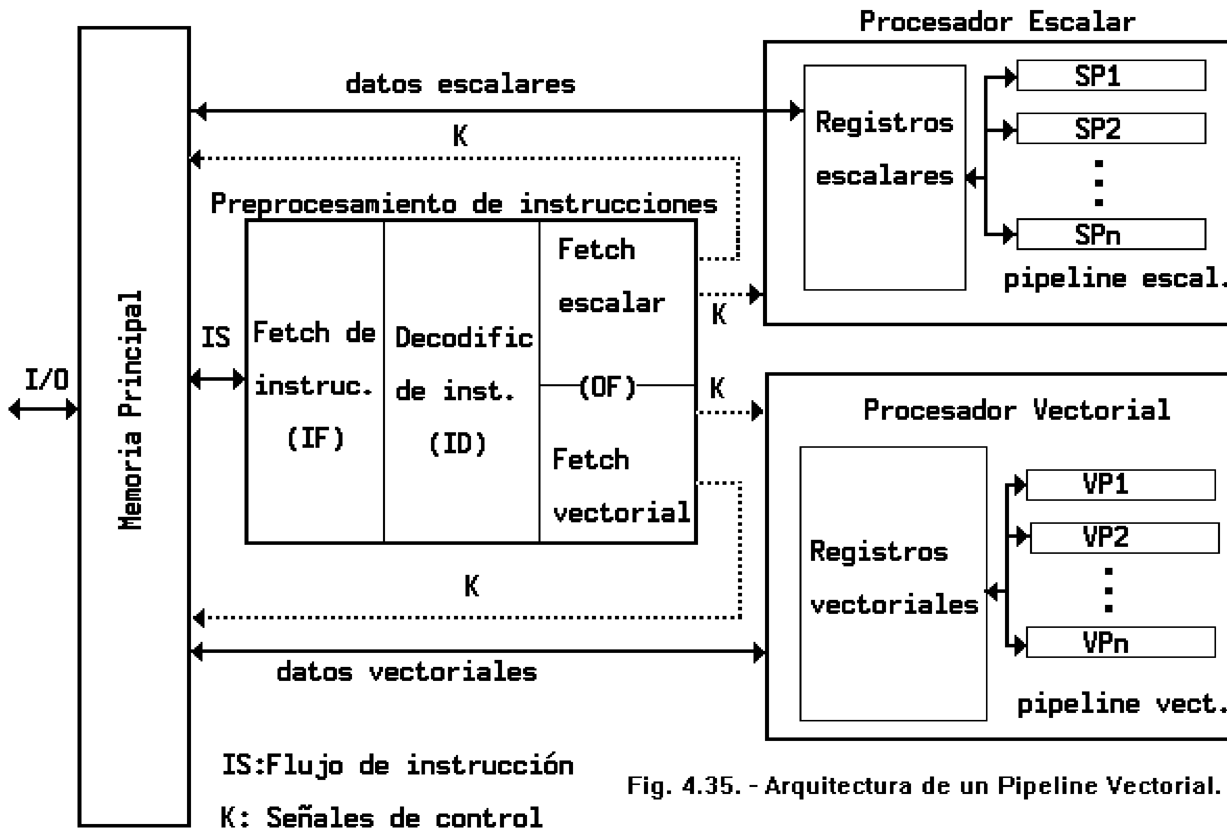


Fig. 4.35. - Arquitectura de un Pipeline Vectorial.

4.11. - PIPELINES VECTORIALES.

Existen aplicaciones que por su gran volumen de información a tratar, no es posible resolverlas por medio de procesadores secuenciales (escalares).

Una manera de resolver esto es dotar al procesador de instrucciones que puedan actuar sobre un gran conjunto de datos, esto se logra por el desarrollo de subrutinas adecuadas o incluyendo en el set de instrucciones del procesador instrucciones capaces de manejar ese volumen de datos. (instrucciones vectoriales que inducen obviamente a una ampliación del microcódigo).

Como ya fue explicado un procesador que maneja instrucciones vectoriales se denomina Procesador Vectorial, los pipelines vectoriales son procesadores vectoriales que tienen una organización interna que combina unidades funcionales múltiples operando en paralelo, junto con procesamiento pipeline dentro de las unidades funcionales.

La arquitectura básica de un procesador Vectorial sería la de la Fig. 4.35.

La funcionalidad de esta máquina puede esquematizarse como se ve en la Fig. 4.36 y opera de la siguiente forma:

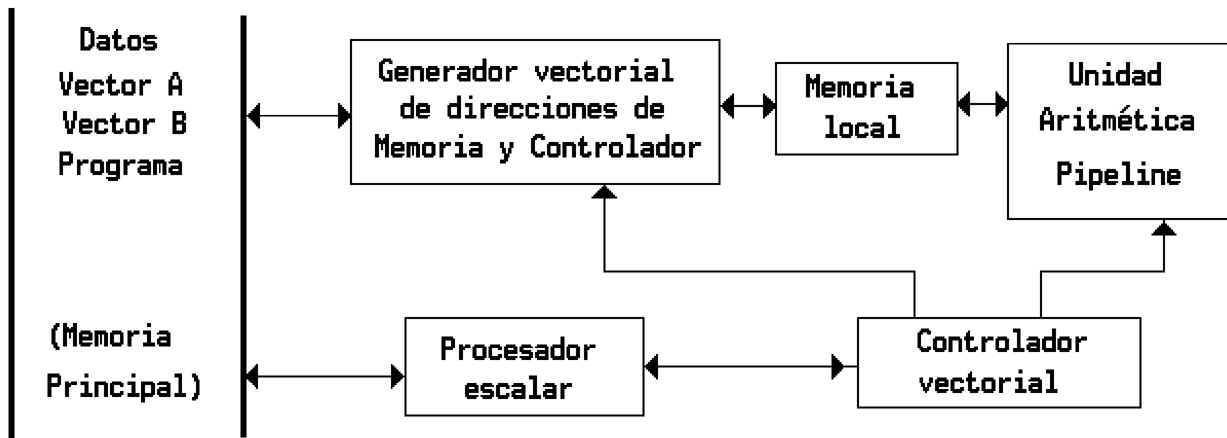


Fig. 4.36. - Funcionalidad de un pipeline vectorial.

El Procesador escalar va ejecutando las instrucciones escalares, hasta que se encuentra con una de tipo vectorial (de manejo de vectores), a partir de ese momento transfiere esa instrucción al controlador vectorial, el cual envía la información de descripción de los vectores (dirección, dimensiones, etc.) al Generador Vectorial y el código de operación a la Unidad Aritmética.

El Generador Vectorial toma la información de Memoria Principal y la coloca en la Memoria local para que esté a disposición de la Unidad Aritmética, la que ejecuta la instrucción deseada según el código de operación enviado por el Controlador Vectorial y es controlada por el mismo.

Nótese, en particular, que en la arquitectura del computador de la Fig. 4.35 la unidad Aritmética del procesador escalar también se encuentra pipelinizada y la unidad de preprocesamiento de las instrucciones también.

Como un ejemplo de una ejecución de instrucción vectorial consideraremos el siguiente ciclo DO:

```
        DO 10  I = 1,50
10      A(I) + B(I)
```

Esto define un circuito de programa que suma conjuntos de números. Para la mayoría de los computadores el lenguaje equivalente de máquina del ciclo DO anterior es una secuencia de instrucciones que leen un par de operandos de los vectores A y B y realizan una suma de punto flotante. Las variables de control del circuito son entonces actualizadas y los pasos repetidos 50 veces.

Un procesador vectorial elimina la cantidad de administración asociada al tiempo que lleva levantar y ejecutar las instrucciones en el circuito de programa. Esto se debe a que una instrucción vectorial incluye la dirección inicial de los operandos, la longitud de los vectores, la operación a realizar y los tipos de los operandos; todos en una instrucción compuesta. Dos operandos A(I) y B(I) son leídos simultáneamente de una memoria interleaved y se envían a un procesador pipeline que realiza la suma de punto flotante. Cuando se obtiene C(I) como salida del pipe se almacena en el tercer módulo interleaved mientras que las operaciones sucesivas se están realizando dentro del pipe.

4.11.1. - Formato de instrucciones vectoriales

El código de operación de una instrucción vectorial equivale a la **selección** de una **tabla de reservación** a utilizar por la unidad aritmética. Las tablas de reservación están implementadas por hardware o microcódigo.

Las instrucciones son suma vectorial, producto interno, producto vectorial, etc. Estas instrucciones deberán tener por lo menos :

- Código de operación
- Dirección de vectores
- Dimensiones de vectores
- Número de elementos de cada dimensión
- Tipo de dato de cada elemento
- Disposición de los elementos en memoria

4.11.2. - **Historia y un ejemplo**

La primer arquitectura de procesador vectorial se desarrolló en los años 60 y más tarde en los 70 para poder realizar directamente cálculos masivos sobre matrices o vectores.
Los procesadores vectoriales se caracterizan por múltiples unidades funcionales pipelinizadas que operan concurrentemente e implementan operaciones aritméticas y booleanas tanto escalares como matriciales.

Tales arquitecturas proveen procesamiento vectorial paralelo haciendo fluir secuencialmente los elementos del vector a través de una unidad funcional pipelinizada y ruteando los resultados de salida de una unidad hacia el input de otra unidad pipeline (proceso conocido como "encadenamiento" -chaining-).

Una arquitectura representativa puede tener una unidad de suma vectorial de seis etapas (Ver Fig. 4.37).

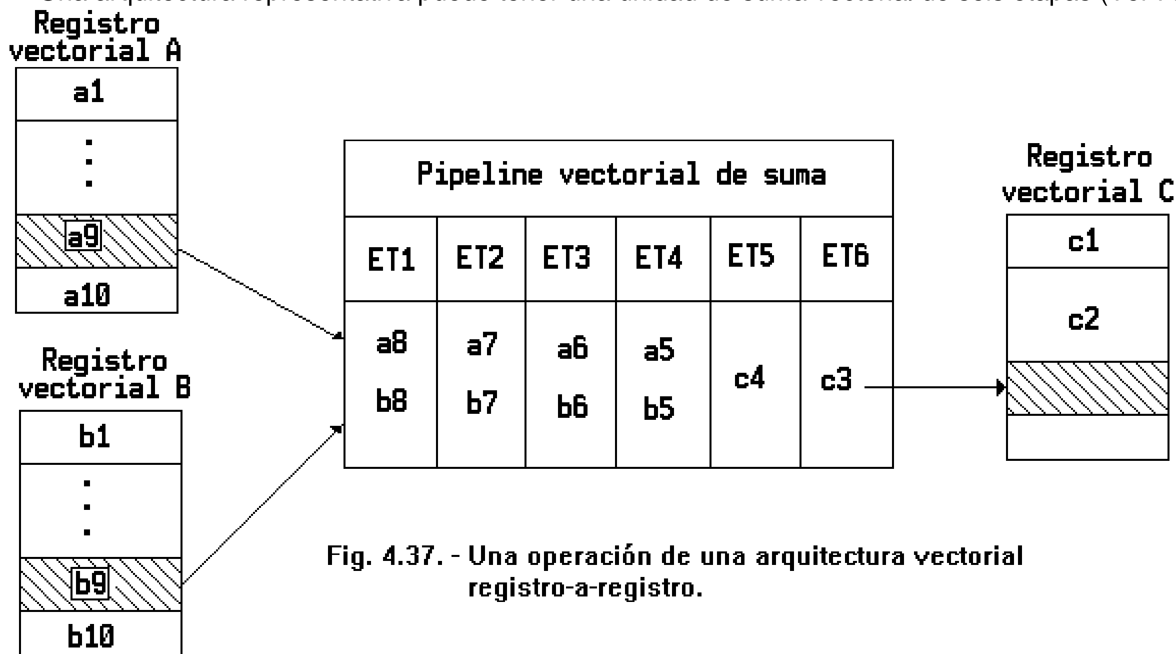


Fig. 4.37. - Una operación de una arquitectura vectorial registro-a-registro.

Si cada etapa del pipe de esta hipotética arquitectura tiene un ciclo temporal de 20 nanosegundos, entonces resultan 120 nanosegundos desde que los operandos **a1** y **b1** ingresan en la etapa 1 hasta que el resultado **c1** se halla disponible.

Cuando el pipe se llena tenemos sin embargo un resultado cada 20 nanosegundos. Luego, el tiempo de carga de las unidades vectoriales pipelinizadas tiene significativas implicancias en cuanto a performance.

En el caso de las arquitecturas registro-a-registro que se muestra en la figura, existen registros vectoriales de alta velocidad para los operandos y resultados.

Se obtiene una performance eficiente en tales arquitecturas (por ej. la Cray-1 y la Fujitsu VP-200) cuando la longitud de los elementos del vector es un múltiplo del tamaño de los registros vectoriales.

Las arquitecturas memoria-a-memoria como las de la Control Data Cyber 205 y la Advanced Scientific Computer de Texas Instruments (ASC) utilizan buffers especiales de memoria en lugar de registros vectoriales.

Las recientes supercomputadoras con procesamiento vectorial (tales como la Cray X-MP/4 y la ETA-10) reúnen entre 4 a 10 procesadores vectoriales mediante una gran memoria compartida.

EJERCICIOS

1) Cómo se logra el paralelismo en los sistemas dentro de la CPU ?
2) En un sistema con una única CPU y sin pipeline puede existir algún tipo de paralelismo ? En caso afirmativo describa un caso concreto.
3) Qué tipos de paralelismo (asincrónico, espacial, temporal) explotan las siguientes arquitecturas : Pipeline, Procesador Array y MIMD ? Justifique.
4) En cuántas etapas puede dividirse la ejecución de una instrucción ? Descríbalas **todas**.
5) Cómo es un pipeline de instrucción ? Grafíquelo.

6) Cuál es la diferencia entre un pipeline aritmético y un Pipeline de Instrucción ?
7) De qué depende la velocidad de un pipe ?
8) Qué es el tiempo de carga de un pipe ? Cómo influye en su rendimiento ?
9) Cuál es la utilidad de los latches ?
10) Defina específicamente :
- Frecuencia de un pipe
- Aceleración
- Eficiencia
- Troughput

11) Cuáles son las diferencias entre Pipeline y Solapamiento ?
12) Enumere la clasificación de Händler de los procesadores pipeline.
13) Diga si es verdadera o falsa la siguiente sentencia :
"Un procesador que cuenta con un coprocesador matemático para las operaciones de punto flotante es una forma de implementación de un pipeline aritmético". Justifique.
14) Cuál es la diferencia entre :
- Pipelines unifuncionales y Multifuncionales.
- Pipelines estáticos y Dinámicos.
- Pipelines escalares y Vectoriales.

15) Diga si son verdaderas o falsas las siguientes sentencias y justifique :
"Un pipe no puro tiene alguna conexión feedback o feedforward".
"Las conexiones feedback no permiten recursividad".
16) Qué es una tabla de reservación y para qué se utiliza ?
17) Construya por lo menos dos tablas de reservación para diferentes funciones que podría realizar el diseño de pipeline del ejemplo (Fig. 4.38).
a)- Construya un diagrama espacio-temporal que muestre la ejecución de ambas funciones, siguiendo la estrategia Greedy, de la secuencia : A A B A A B A A .....
Suponer que el pipeline es :
i) estático
ii) dinámico
18) Construya una única tabla de reservación que muestre la ejecución de ambas funciones del ej.) 17) cuya latencia sea mínima. La latencia elegida resultó ser la latencia greedy ? Porqué ?

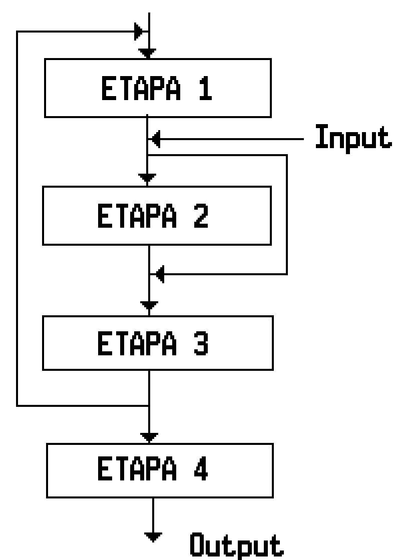


Fig. 4.38.

19)- Sobre la ejecución del siguiente conjunto de instrucciones :
(1) B = A + 1
(2) C = D + 1
(3) Z = B + H
explique cuáles riesgos se pueden suceder (RAW, WAR o WAW) y justifíquelo en base a las intersecciones de los conjuntos R(i) y D(i).
20)- Cuál es el problema que plantean las instrucciones de salto en estructuras pipeline ?
21)- Qué problema se puede suscitar si el ancho de banda de memoria es diferente del necesario para poner a disposición del pipe las instrucciones requeridas ? Discuta el caso en que sea mayor y el caso en que sea menor.
22) Cómo funciona un procesador con pipeline aritmético (vector processing) ?
23) Cuál es la información que se agrega a una instrucción escalar para transformarla en una instrucción vectorial ?
24) Cuáles son las condiciones para poder aplicar un pipe ?
25) El flujo de datos dentro de un pipe es discreto o continuo ?
26) Quién sincroniza el flujo de datos dentro de un pipe ?
27) Cómo funciona un procesador vectorial pipelinizado ?

CAPITULO 5

ARQUITECTURAS SIMD

5. - Generalidades

En una arquitectura SIMD el paralelismo se logra por múltiples unidades de proceso llamadas Elementos de Procesamiento (PEs), cada una de las cuales es capaz de ejecutar una operación especializada autónomamente.

La arquitectura de los Array Processor y de los Procesadores Vectoriales SIMD se caracteriza por el hecho de que la misma operación es realizada en un momento dado sobre un gran conjunto de datos en todos los PEs.

Las computadoras SIMD están especialmente diseñadas para realizar cómputos vectoriales sobre matrices o arrays de datos.

Pueden utilizarse indistintamente los términos Procesadores Array, Procesadores Paralelos, y computadoras SIMD.

5.1 - Array Processor.

Un array sincrónico de procesadores paralelos se denomina un Procesador Array, el cual consiste de múltiples Elementos de Procesamiento (PEs) bajo la supervisión de una unidad de control (CU).

Un procesador array puede manejar un único flujo de instrucción y múltiples flujos de datos. En tal sentido los procesadores array son también conocidos como máquinas SIMD (Fig. 5.1).

Las procesadores Array existen en dos organizaciones arquitecturales básicas :

- Los procesadores array que utilizan memoria de acceso aleatorio
- Los procesadores asociativos que usan memorias direccionables por contenido (o memorias asociativas).

Trataremos los procesadores array y brevemente los de memoria asociativa.

Dado que un PE no constituye una unidad de proceso central completa (CPU) y que no es capaz de funcionar independientemente, el sistema es en este caso de procesamiento paralelo y no un sistema multiprocesador.

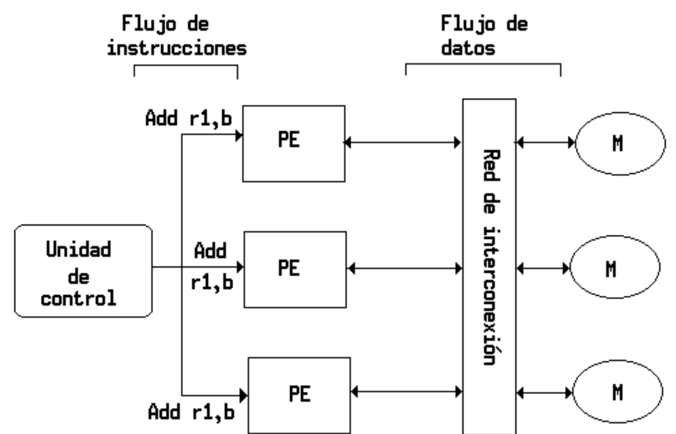


Fig. 5.1. - Ejecución SIMD.

5.2 - Organización de las computadoras SIMD

En general, un procesador array puede tener una de dos configuraciones ligeramente diferentes.

Una ha sido implementada en la muy conocida computadora ILLIAC IV cuya esquematización puede verse en la Fig. 5.2. Esta computadora posee 256 PEs distribuidos en cuatro cuadrantes de 8 x 8 PEs (Fig. 5.3).

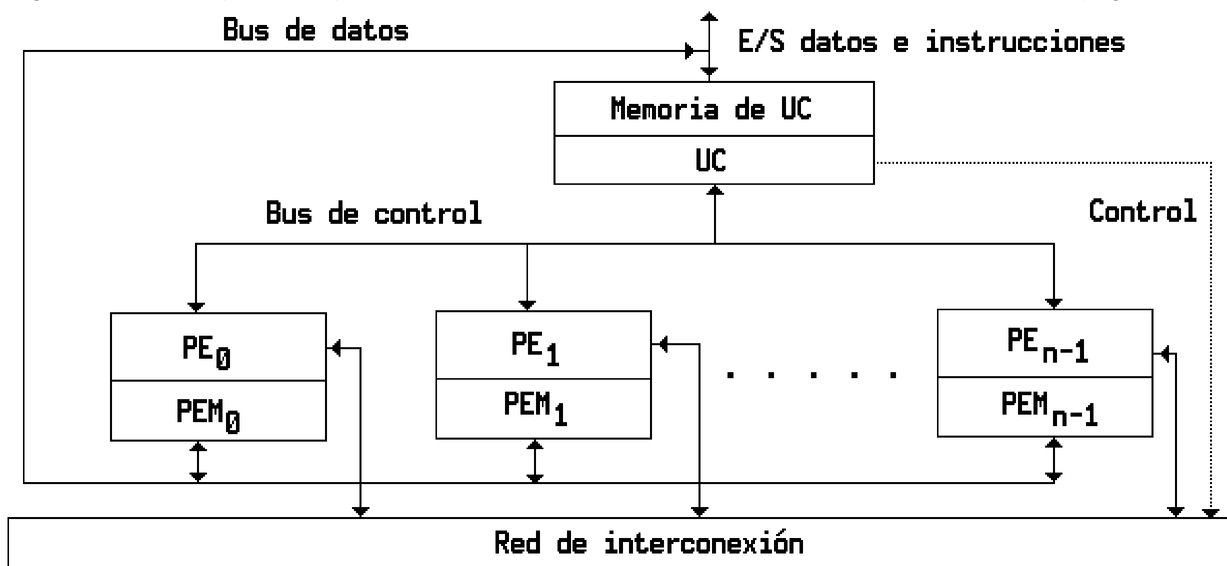


Fig. 5.2. - Un tipo de configuración de un procesador Array (ILLIAC IV).

Los PEs de un cuadrante son controlados por una unidad de control común, como resultado de esto los PEs de dicho cuadrante pueden ejecutar la misma operación simultáneamente.

La Fig. 5.3 muestra la comunicación entre los PEs dentro del cuadrante. Cada PE se comunica con su vecino, la mayor distancia entre dos PEs no vecinos es 8.

La esquematización de la ILLIAC IV de la Fig. 5.2 está estructurada con N PEs sincronizados, los que se encuentran bajo el control de una CU.

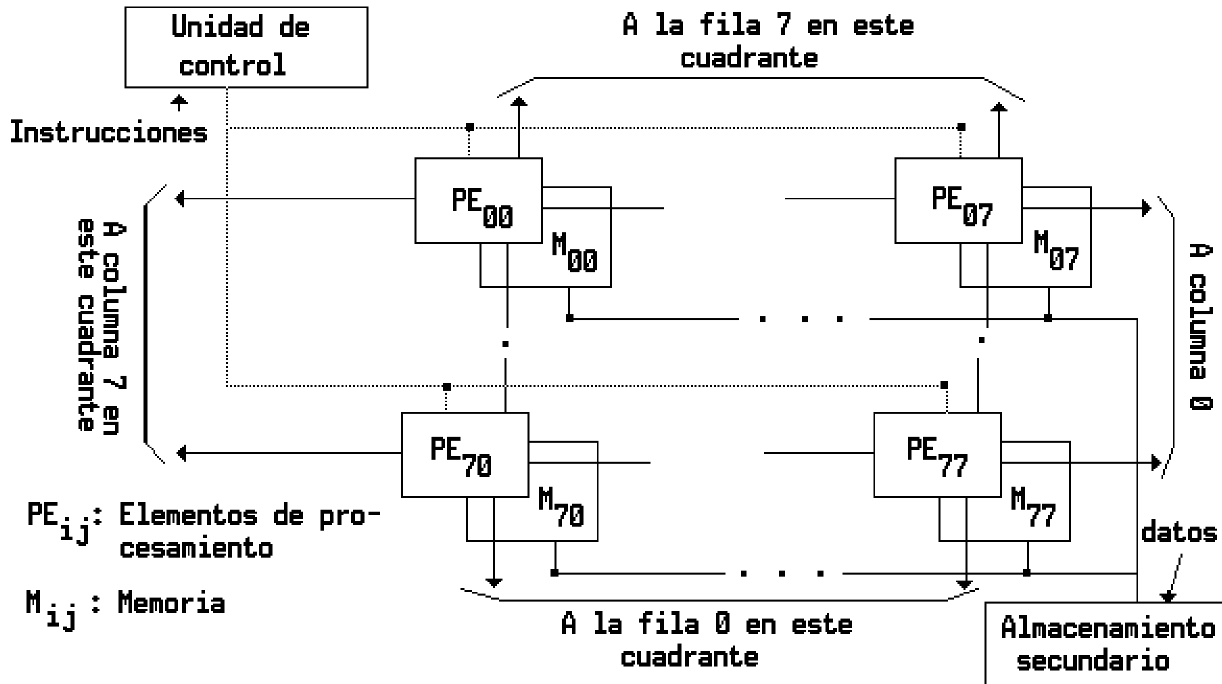


Fig. 5.3. - Cuadrante de 8 * 8 PEs de la ILLIAC IV.

Cada PE es esencialmente una unidad aritmético y lógica (ALU) con registros de trabajo que le pertenecen y una memoria local (PEM) para el almacenamiento de los datos distribuidos.

La CU cuenta también con una memoria propia para el almacenamiento de programas. En ella se cargan los programas del usuario desde el almacenamiento principal.

La función de la CU es decodificar todas las instrucciones y determinar dónde deben ejecutarse las instrucciones decodificadas.

Las instrucciones de control de tipos o las escalares se ejecutan directamente dentro de la CU. Las instrucciones vectoriales se envían a los PEs para una distribución de la ejecución a fin de alcanzar un paralelismo espacial gracias a la multiplicación de unidades aritméticas (los PEs).

Todos los PEs realizan la misma función sincrónicamente en forma esclava bajo el control de la CU.

Los operandos vectoriales se distribuyen en las PEM antes de lanzar la ejecución paralela de los PEs.

Estos datos distribuidos pueden cargarse en las PEM desde una fuente externa a través del bus de datos del sistema, o desde la CU en una modalidad que utiliza el bus de control.

La unidad de control emite y distribuye máscaras sobre los PEs para permitir la realización o no de ciertas operaciones, para ello lleva cuenta del estado de los PEs utilizando bits de estado asociados a los mismos.

Un array processor facilita la programación brindando operaciones sobre vectores y escalares; es posible ejecutar una operación de instrucción sobre un vector completo, con las únicas restricciones del tamaño de memoria y del número de PEs disponibles.

Se utilizan esquemas de enmascaramiento para controlar el estado de cada PE durante la ejecución de la instrucción vectorial.

Cada PE puede estar en estado activo o en estado inhibido durante el ciclo de la instrucción indicado por un vector de máscaras.

En otras palabras, no es necesario que todos los PEs participen en la ejecución de una instrucción vectorial, sólo los PEs habilitados realizan el cómputo.

Los intercambios de datos entre los PEs se realizan vía una red de comunicación inter-PE. Esta red de comunicación está bajo el control de la CU.

Un array processor tiene normalmente una interfase a un computador host desde su unidad de control. El computador host es una máquina de propósito general que sirve como el "administrador operativo" del sistema en su totalidad, formado por el host y el procesador array.

Las funciones del host consisten en la administración de los recursos y la supervisión de los periféricos y las entradas/salidas.

La unidad de control del procesador array supervisa directamente la ejecución de los programas, en tanto que el host realiza las funciones ejecutivas y de E/S con el exterior. En tal sentido un procesador array puede ser

considerado como un computador conectado back-end (attached), similar en cuanto a función a los procesadores pipeline attachados.

Otra configuración posible para un procesador array se ilustra en la figura 5.4. Esta difiere de la de la Fig. 5.2 en dos aspectos.

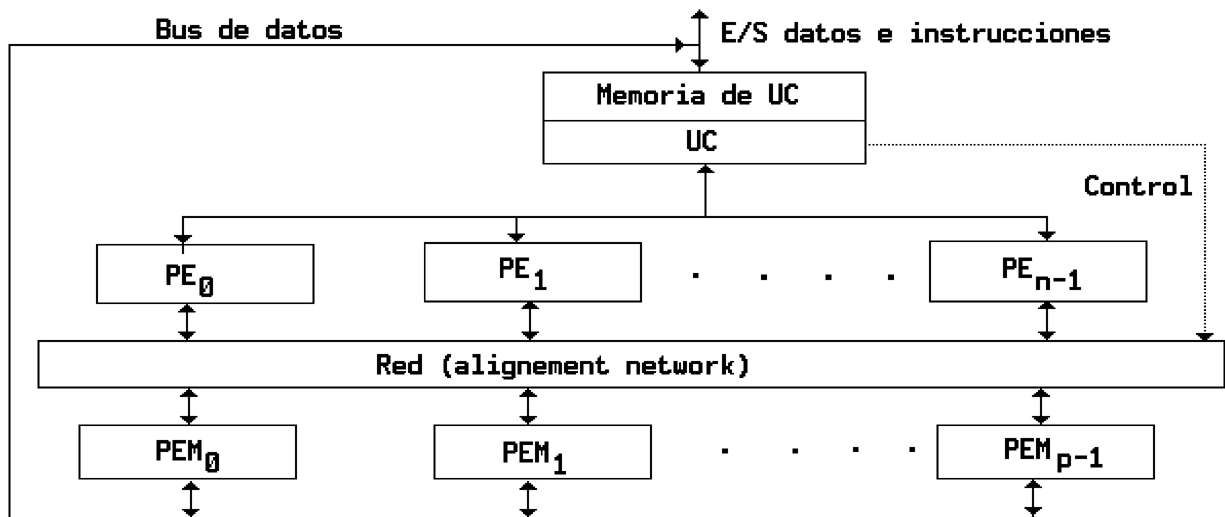


Fig. 5.4. - Otro tipo de configuración de un procesador Array (computador BSP de Burroughs).

Primero, las memorias locales pertenecientes a cada PE se reemplazan aquí por módulos paralelos de memoria compartidos por todos los PEs mediante una red (alignement network).

Segundo, la red de permutación de PEs se reemplaza por una red inter-PE de memoria, que se encuentra nuevamente bajo control de la CU.

Esta red (alignement) es una red de caminos conmutables entre los PEs y las memorias paralelas. Se espera que tal red sirva para liberar de los conflictos de acceso a las memorias compartidas por tantos PEs como sea posible. Un buen ejemplo de esta configuración es el Procesador Científico Burroughs (BSP).

En este tipo de configuración existen n PEs y p módulos de memoria. Ambos números no son necesariamente iguales.

5.3 - **Estructura interna de un PE**

En la Fig. 5.5 graficamos la estructura interna de un elemento de procesamiento :

- D_i indica la dirección asignada a este elemento de procesamiento.
- S_i indica si este procesador está o no activo para realizar un cierto cálculo.
- I_i es un registro índice.
- R_i es el registro que apunta a otros procesadores

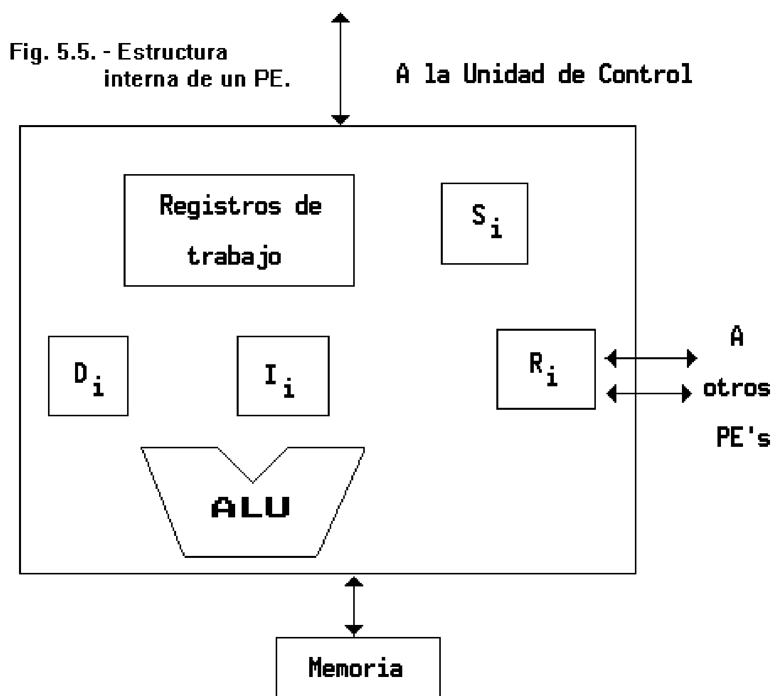
Formalmente un procesador SIMD se caracteriza por el siguiente conjunto de parámetros :

C = { N, F, I, M }

siendo

- N = el número de procesadores.
- F = el conjunto de funciones de ruteo de datos previstos por la red de interconexión.
- I = el conjunto de instrucciones para las operaciones escalares vectoriales, de ruteo de datos y manejo de la red.
- y M = conjunto de máscaras que habilitan o deshabilitan a los procesadores.

Fig. 5.5. - Estructura interna de un PE.



5.4 - **Ejemplo de funcionamiento de un procesador array**

Para mostrar cómo es el ruteo de datos en un procesador array veremos en detalle una instrucción vectorial dentro de los N PEs.

Se desea la suma de S(k) de las primeras k componentes de un vector A para k = 0,1,....,n-1.

Sea A = (A0, A1, ..., An-1)

Necesitamos calcular las siguientes n sumas:

Este vector de n sumas puede calcularse recursivamente realizando las n-1 iteraciones definidas como :

S(0) = A(0)

S(k) = S(k-1) + A(k)   para k = 1,2,.....,n-1

Estas sumas recursivas para n = 8 se implementaron en un procesador array N = 8 PEs en una cantidad de pasos de [$\log_2$ n]=3.

En la implementación se utilizan tanto el ruteo de datos como el enmascaramiento de PEs.

Inicialmente cada A(i) que reside en cada PEM(i) es movido al registro D(i) en cada PE(i) para i = 0,1,....,n-1 (asumimos aquí n=N=8).

En el primer paso A(i) es ruteado desde R(i) a R(i+1) y sumado al A(i+1), obteniéndose la suma A(i) + A(i+1) en R(i+1) para i=0,1,....,6.

Las flechas en la Fig. 5.6 muestran las operaciones de ruteo y la notación abreviada Σ A_iaA_j se utiliza indicando la suma intermedia de A(i) + A(i+1)+ ... +A(j).

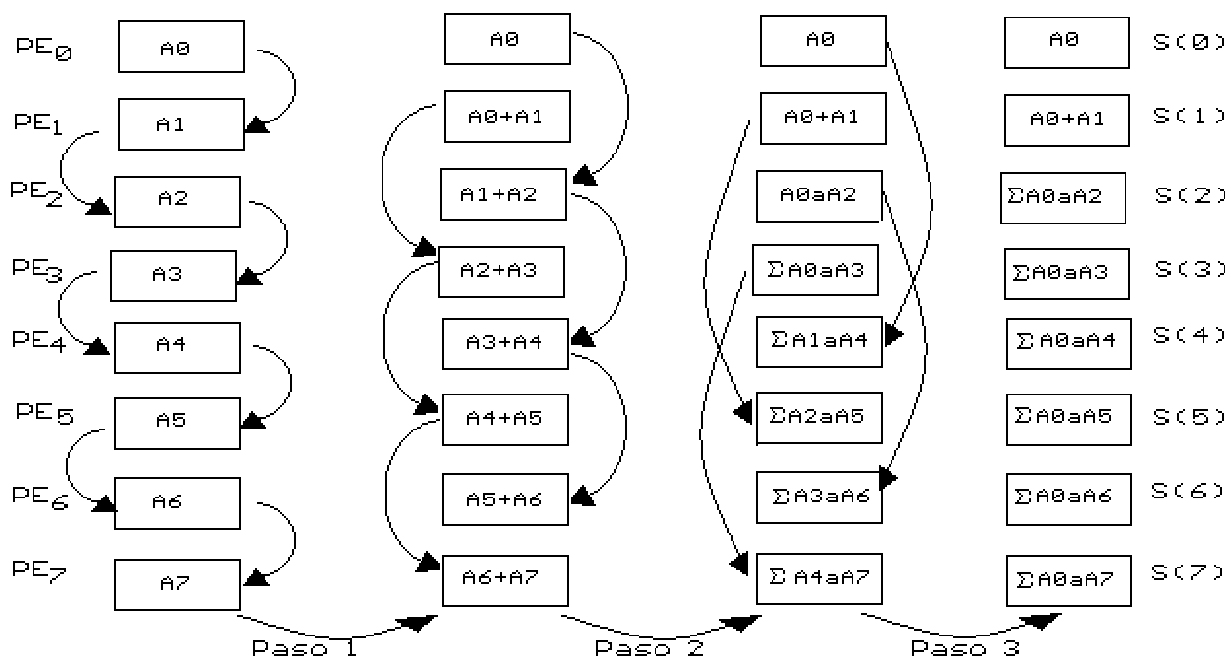


Fig. 5.6. - Cálculo de la suma S(k) = Σ A(k), para k = 0,...,7 en una máquina SIMD.

En el paso 2, las sumas intermedias en R(i) son ruteadas a R(i+2) para i = 0,1,....,5.

En el paso final las sumas intermedias en R(i) son ruteadas a R(i+4) para i = 0,..,3

Consiguientemente, el PE(k) tiene el valor final de S(k) para k = 0,1,...,7.

Cuanto más lejos se extienden las operaciones de ruteo cada vez menos PEs son los intervinientes. Nótese que en el paso 1 PE(7) recibe pero no transmite; en el paso 2 son los PE(6) y PE(7); en tanto que en el paso 3 son los PE(4), PE(5), PE(6) y PE(7) los que reciben solamente.

Estos PEs que no se desea que transmitan son enmascarados durante el paso correspondiente.

Durante las operaciones de suma PE(0) permanece inhibido en el paso 1; PE(0) y PE(1) se vuelven inactivos en el paso 2; en tanto que en el paso 3 los PE(0), PE(1), PE(2) y PE(3) son inactivos.

Los PEs que son enmascarados en cada paso dependen de la operación (ruteo de datos o suma aritmética) que se realice.

Aún así los patrones de enmascaramiento varían en los diferentes ciclos de la operación.

Nótese que las operaciones de ruteo y enmascaramiento pueden ser mucho más complejas cuando la longitud del vector es $n \geqslant N$.

Un array de PEs es una unidad aritmética pasiva esperando ser invocada para las tareas de cómputo paralelo.

5.5. - **Procesadores Array de estructura Bit-Plane**

Una variante de las arquitecturas de procesadores array utiliza una gran cantidad de procesadores de un bit. En las arquitecturas **bit-plane** los procesadores array se colocan en una grilla (mesh) simétrica (por ej. de 64 * 64) y se asocian con múltiples planos de bits en memoria que se corresponden a las dimensiones de la grilla de procesadores (ver Fig. 5.7).

El procesador n (Pn) situado en la grilla en la posición (x,y) opera sobre los bits de memoria ubicados en la posición (x,y) en todos los planos de memoria asociados.

Usualmente se proveen además operaciones del tipo de copia, enmascarado y operaciones aritméticas que operan sobre todos los planos de memoria tanto sobre columnas como sobre filas.

El Procesador Masivo Paralelo de Loral y el Procesador Array Distribuido de ICL ejemplifican esta clase de arquitecturas utilizadas muy frecuentemente para aplicaciones de procesamiento de imágenes mapeando pixels desde la estructura planar de la memoria.

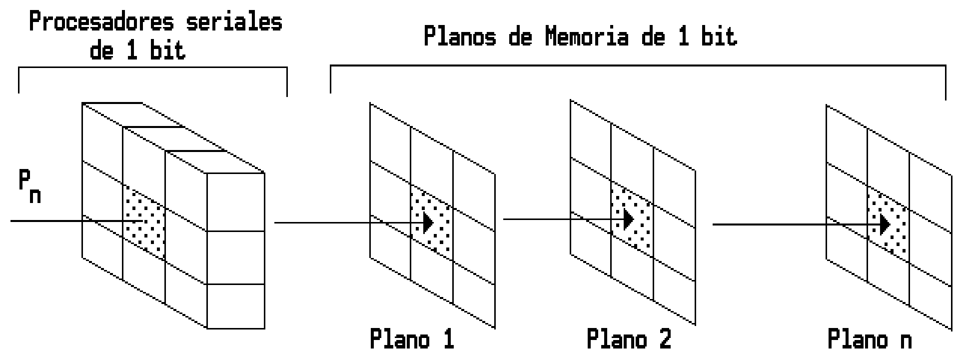


Fig. 5.7. - Procesador array de estructura bit-plane.

La Máquina de Conexión de Thinking Machine tiene organizada la memoria como 65.536 procesadores de un bit en conjuntos de a cuatro procesadores unidos en una topología de hipercubo.

5.6. - ARQUITECTURA DE PROCESADORES ARRAY DE MEMORIA ASOCIATIVA

Las computadoras construidas sobre memorias asociativas constituyen un tipo muy distintivo de máquinas SIMD que utilizan una lógica de comparación para acceder en paralelo, de acuerdo al contenido, a los datos almacenados.

Las investigaciones sobre la construcción de memorias asociativas comenzaron en los últimos años de la década de los 50 con el objetivo obvio de lograr recorrer la memoria en paralelo buscando datos que coincidieran con una clave específica.

Los procesadores de memorias asociativas "modernos" desarrollados en los 70 (por ej. el Conjunto de Elementos de Procesamiento en Paralelo -PEPE- de los laboratorios Bell) y las arquitecturas más recientes (el procesador Asociativo de Loral, ASPRO) han sido construidos lógicamente orientados a aplicaciones de bases de datos tales como búsquedas y navegaciones.

La Fig. 5.8 muestra las características de las unidades funcionales de un procesador de memoria asociativa. Un controlador de programa (un computador serial) lee y ejecuta las instrucciones invocando a un controlador array especializado cuando se detectan instrucciones de memoria asociativa.

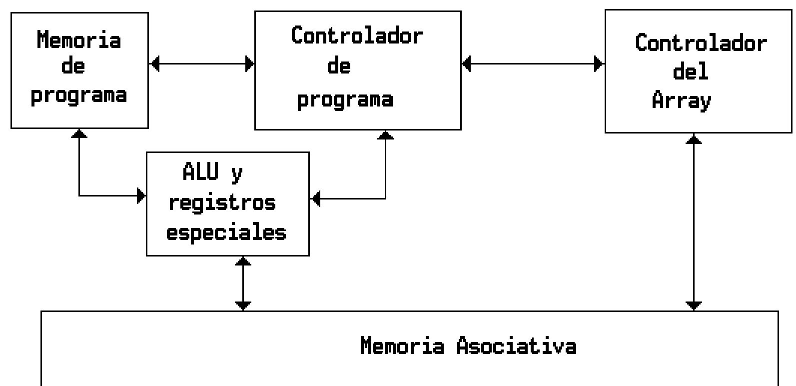
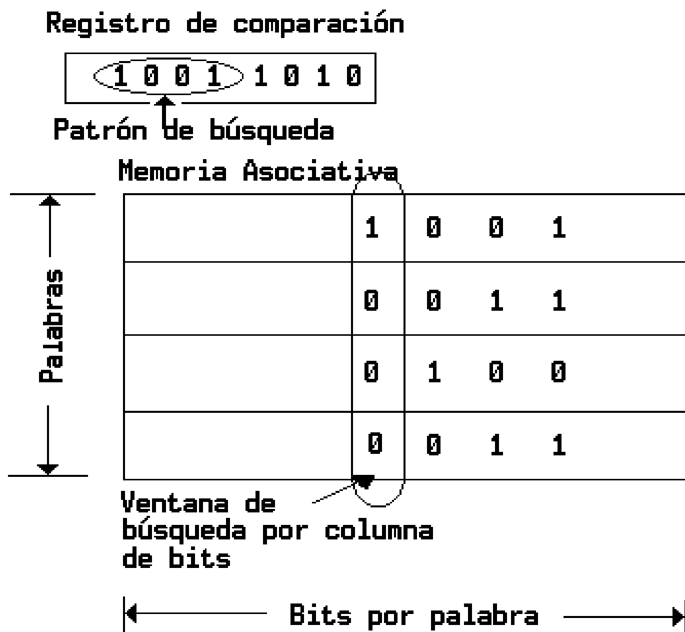


Fig. 5.8. - Organización de un procesador de memoria asociativa.



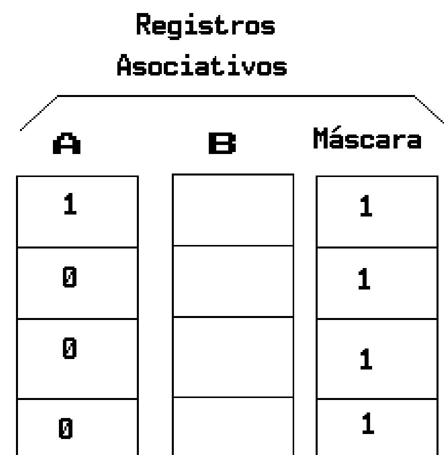


Fig. 5.9. - Comparación orientada a fila.

Registros especiales habilitan al controlador de programa y a la memoria asociativa para compartir datos.

La mayoría de los procesadores de memoria asociativa utilizan una organización serial de bits, la cual implica operaciones concurrentes sobre un solo bit-slice para todas las palabras en la memoria asociativa.

Cada palabra de memoria asociativa, que generalmente tiene una gran cantidad de bits (por ej. 32768) se asocia con registros especiales y una lógica de comparación que funcionalmente constituye un procesador. Por lo tanto, un procesador asociativo con 4096 palabras tiene efectivamente 4096 elementos de procesamiento.

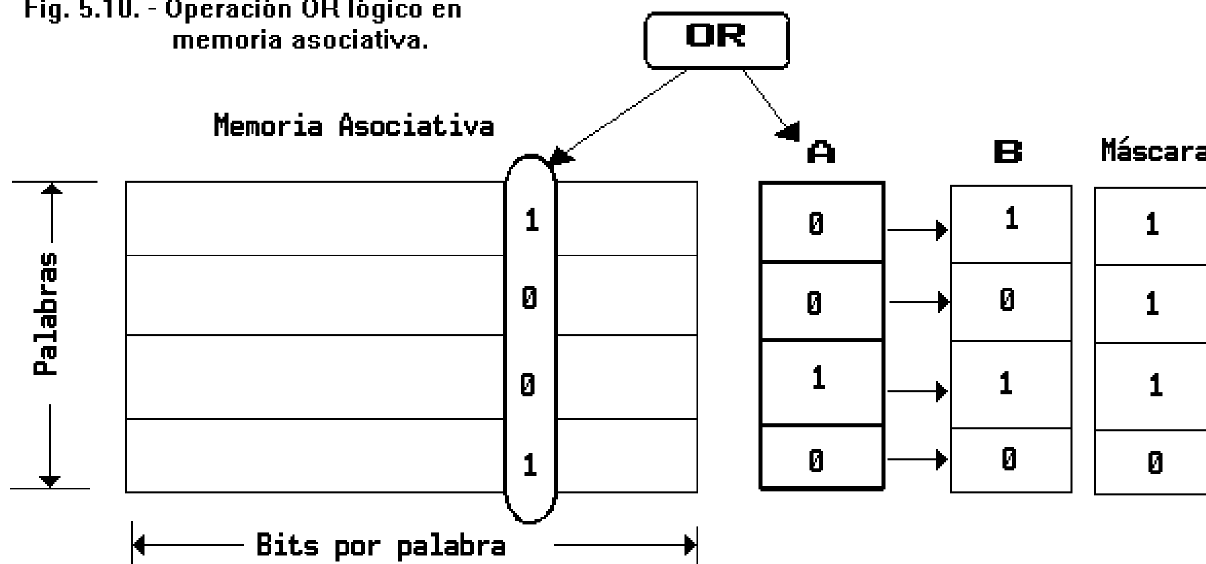
La Fig. 5.9 muestra una comparación orientada a fila para una arquitectura genérica de bit-serial. Una parte del registro de comparación contiene el valor que debe coincidir.

Todos los elementos de procesamiento (PE) comienzan en una columna específica de la memoria y comparan el contenido de cuatro bits consecutivos en la fila con el contenido del registro, encendiendo un bit en el registro A para indicar si la fila equipara o no.

En la Fig. 5.10 podemos ver una operación de un OR lógico realizada entre una columna de bits y el vector de bits en el registro A, siendo el registro B el que recibe el resultado.

Un cero en el registro de máscara indica que la palabra asociada no debe incluirse en esta operación.

Fig. 5.10. - Operación OR lógico en memoria asociativa.



5.7.- **SYSTOLIC ARRAYS (SISTÓLICOS)**

Los procesadores sistólicos son el resultado de los avances en tecnología de semiconductores y en las aplicaciones que requieren un amplio rendimiento.

5.7.1 - **Introducción a los procesadores Sistólicos**

Fue en 1978 cuando H. T. Kung y C. E. Leiserson introdujeron el término "sistólico" y el concepto subyacente, para resolver problemas de sistemas de propósito específico que deben balancear el bandwidth entre una intensiva cantidad de cálculos y gran cantidad de requerimientos de E/S.

Los procesadores convencionales están muy a menudo limitados por la disparidad del bandwidth de Entrada y el bandwidth de Salida, el cual se produce debido a que los ítems de datos deben ser leídos y grabados cada vez que se los referencia.

Una razón para elegir el término "systolic" como parte de Systolic Array puede verse en la analogía con el sistema de circulación humano, en el cual el corazón entrega y recibe una gran cantidad de sangre como resultado del bombeo rítmico e ininterrumpido de pequeñas cantidades de ese fluido a través de venas y arterias. En esta analogía el corazón corresponde a la fuente y destino de los datos, como si fuera una memoria global; y la red de venas es equivalente al array de procesadores y sus conexiones.

Las arquitecturas Sistólicas (array sistólicos) son *multiprocesadores pipelinizados* en los cuales los datos se bombean en forma rítmica desde la memoria y a través de la red de procesadores antes de ser devueltos a la memoria (ver Fig. 5.11).

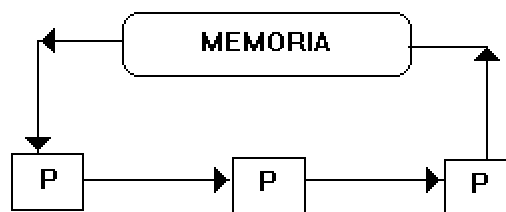


Fig. 5.11. - Flujo de datos desde y hacia la memoria.

La información circula entre los procesadores como en un pipeline, pero sólo los procesadores frontera mantienen comunicación con el exterior.

Un reloj global conjuntamente con mecanismos explícitos de retardo *sincronizan* el flujo de datos a través del pipe que se conforma con los datos obtenidos de la memoria y los resultados parciales que usa cada procesador.

Los procesadores modulares unidos mediante una red local y regular proveen los ladrillos básicos para construir una buena variedad de sistemas de propósito específico.

Durante cada intervalo de tiempo estos procesadores ejecutan una secuencia corta e invariante de instrucciones.

El término "array" se origina en la similitud de los systolic array con una grilla o red en la cual cada punto corresponde a un procesador y cada hilo a una conexión entre los procesadores. Visto como esta estructura los sistólicos son descendientes de las arquitecturas del tipo array, tales como los arrays interactivos, los autómatas celulares y los procesadores array.

Si bien la estructura array caracteriza las interconexiones en los sistólicos, es el término "systolic" el que capta el comportamiento innovador y distintivo de estos sistemas.

"Systolic" en este contexto significa que los cálculos pipeline se realizan en todas las dimensiones del array y brindan como resultado un muy alto rendimiento computacional. Son sistemas de cómputos altamente especializados en los que se debe realizar operaciones concurrentes y que se caracterizan por la gran cantidad y diversidad de procesadores.

La tecnología apta para esto es la VLSI (Integración en muy alta escala). De todas maneras existe un límite tecnológico, inclusive el de la velocidad de la luz. Luego para conseguir altas velocidades de procesamiento se recurre al uso simultáneo de procesadores. O sea que los algoritmos a utilizar deben permitir un alto grado de pipelining y multiprocesamiento.

En otras palabras, los algoritmos sistólicos administran los cálculos de manera tal que un ítem de dato no se usa solamente cuando es input sino también es *reutilizado moviéndose a través del pipeline en el array.*

En estos casos se hacen críticas las redes de interconexión. La misma consiste en un conjunto de procesadores interconectados que realizan operaciones simples. La interconexión puede generar vectores, matrices, árboles, etc.

Esto resulta en un balance del ancho de banda (bandwidth) entre el procesamiento y la entrada/salida, especialmente en problemas de compute-bound que tienen más cálculos a realizar que entradas y salidas.

La idea (como se ve en la Fig. 5.11) es que desde la memoria se "bombea" la información como desde un corazón, y esta fluye de un Procesador al siguiente.

Esto es adecuado para procesos con mucho tiempo de cálculo y donde varias operaciones se realizan en forma repetida sobre un mismo dato.

Fig. 5.12. - Aplicaciones sobre procesadores Sistólicos.

* Procesamiento de señales/imágenes y reconocimiento de patrones
  - Filtros digitales
  - Figuras en dos dimensiones
  - Transformadas discretas de Fourier
  - Interpolación
  - Alabeo geométrico
  - Lineamientos de extracción
  - Estadísticas de orden
  - Clasificación de distancia-mínima
  - Cálculo de covarianza de matrices
  - Coincidencia de patrones (pattern matching)
  - Reconocimiento de patrones sintácticos
  - Procesamiento de señales de radar
  - Detección de curvas
  - Animación de figuras
  - Comparación de imágenes
* Aritmética de matrices
  - Multiplicación y triangulación de matrices
  - Descomposición QR
  - Operaciones sobre matrices dispersas
  - Solución de sistemas lineares triangulares
* Aplicaciones No-numéricas
  - Estructuras de datos - stacks y colas, ordenamiento
  - Algoritmos gráficos - clausura transitiva, minimización de árboles de máxima dimensión
  - Conexión de componentes
  - Reconocimiento del lenguaje
  - Programación dinámica
  - Operaciones sobre bases de datos relacionales
  - Aritmética sobre arrays

Este tipo de arquitecturas es la apropiada para resolver problemas de alta repetición y muy específicos, o sea se desarrollan para casos especiales y no se busca la resolución de problemas en general.

5.7.2. - **Características de los Procesadores Sistólicos.**

La Fig. 5.12 es una lista de las aplicaciones disponibles sobre diseños sistólicos.

Es apropiado hablar brevemente sobre los tres factores que caracterizan a los sistólicos así como fueron originalmente propuestos, a saber: tecnología, procesamiento pipeline/paralelo, y aplicaciones.

Estos factores identifican también las razones para el éxito del concepto, especialmente eficacia de costos, alta performance, y la abundancia de aplicaciones para las cuales son utilizados.

5.7.2.1 - **Tecnología y eficacia de costos**

Hoy en día la madurez de la tecnología VLSI/WSI (Wafer Scale Integration) permite fabricar circuitos cuyas dimensiones mínimas rondan los 1 a 3 micrones. La exactitud en el campo de los procesos de fabricación VLSI/WSI hace posible la implementación de circuitos de hasta 1/2 millón de transistores a un costo razonable (incluso para pequeñas cantidades de producción).

Sin embargo las ventajas de esta tecnología no pueden utilizarse cabalmente a menos que se utilicen diseños simples, regulares y modulares. Los sistólicos tratan de cumplir estas restricciones topológicas mediante elementos de procesamiento sencillos, que, conjuntamente con un patrón de interconexión, también sencillo, se distribuyen en una o más dimensiones.

El costo, la regularidad, y la modularidad son factores que influyen en el diseño y la optimización de cada elemento de procesamiento individualmente y en el de sus respectivas interconexiones.

La consideración de estos tres factores indican que los procesadores arrays son una solución de un costo efectivo al problema de la construcción de sistemas con muchos elementos de procesamiento.

Debido a que los sistólicos tienen una gran cantidad de módulos iguales, salvo algunas pocas excepciones, el proceso de refinamiento sobre un gran sistema y el posterior diseño de cada subcomponente es más veloz y sencillo de lo que resulta en sistemas con una gran variedad de tipos de módulos.

5.7.2.2 - **Procesamiento pipeline/paralelo**

La eficacia en los cálculos de los procesadores sistólicos deriva del *multiprocesamiento y la técnica pipeline.* El multiprocesamiento es una consecuencia natural de las actividades que están viajando simultáneamente en distintos elementos de procesamiento de la red. El pipeline puede pensarse como una forma de multiprocesamiento que optimiza la utilización de recursos y saca ventaja de las dependencias entre los cálculos.

En los procesadores sistólicos, el pipeline de datos reduce el bandwidth de E/S permitiendo que un ítem de dato sea reutilizado una vez que ingresó a la red. Típicamente, los inputs entran en la red a través de elementos de procesamiento periféricos y se propagan a los elementos de procesamiento vecinos para ulteriores procesos.

Estos movimientos de datos a través de la red se producen en una *dirección fija* en la cual existe una conexión entre elementos de procesamientos vecinos y además se producen en forma periódica.

Además del pipeline de datos, los procesadores sistólicos se caracterizan también por un *pipeline de cómputo, en el cual la información fluye de un elemento de procesamiento hacia otro según un orden preestablecido.* Esta información puede ser interpretada por el receptor tanto como un dato, o información de control, o una combinación de ambos.

La ejecución procede de manera tal que el output generado por un elemento de procesamiento es utilizado como input de un elemento de procesamiento vecino.

Mientras que las operaciones se suceden como un flujo de datos a través de cada procesador, el cálculo completo no es una operatoria de flujo de datos (Dataflow), ya que las operaciones se ejecutan de acuerdo a un itinerario determinado por el diseño del array sistólico.

Luego de que un elemento de procesamiento genera un output intermedio y lo entrega al elemento de procesamiento vecino, continúa calculando otro output intermedio. Como resultado de esto los recursos de procesamiento se utilizan eficientemente.

Generalmente cada elemento de procesamiento puede estar construido como un procesador pipeline. Tal construcción recibe el nombre de Array Sistólico Pipelinizado de Dos Niveles y brinda un mayor troughput.

5.7.2.3 - **Aplicaciones y algoritmos**

Los algoritmos adecuados para las implementaciones de arrays sistólicos pueden hallarse en gran variedad de aplicaciones, tales como el procesamiento digital de señales e imágenes, el álgebra lineal, el reconocimiento de patrones, programación lineal y dinámica y problemas de grafos. De hecho, muchos de los algoritmos en la lista de aplicaciones contienen gran cantidad de cálculos y necesitan de las arquitecturas sistólicas para poder ser implementados cuando se utilizan en entornos de tiempo real.

5.7.3 - **CUESTIONES DE IMPLEMENTACIÓN**

5.7.3.1 - **Sistemas sistólicos de propósito general y específico**

En forma típica, un array sistólico puede pensarse como un sistema algorítmicamente especializado en el sentido en que su diseño refleja necesidades de un algoritmo específico.

Sin embargo, sería deseable diseñar arrays sistólicos capaces de ejecutar eficientemente más de un algoritmo para una o más aplicaciones.

Existen dos propuestas posibles para el diseño de estos sistemas "multipropósito", y existe un compromiso entre las dos que se encuentra generalmente en muchas de las implementaciones actuales.

Una propuesta consiste en agregar mecanismos de hardware para poder reconfigurar la topología y el patrón de interconexiones del sistólico y poder así, emular los requerimientos de un diseño especializado.

Un ejemplo concreto de esto lo constituye la computadora Configurable de Alto Paralelismo (Configurable Highly Parallel, CHiP), que cuenta con una matriz de switches programable con el propósito de reconfiguraciones.

La otra propuesta utiliza software para mapear diferentes algoritmos dentro de una arquitectura fija de array. Esta propuesta necesita de la existencia de lenguajes de programación capaces de expresar cálculos en paralelo, así como del desarrollo de traductores, sistemas operativos y ayudas de programación.

Estos requerimientos son de aplicación, por ejemplo, en el sistema sistólico WARP desarrollado en la Universidad de Carnegie Mellon.

Para cada algoritmo, el diseñador necesita identificar el diseño sistólico eficiente y los mapas y las técnicas apropiadas para utilizar. La cuestión de las técnicas apropiadas es de gran importancia, ya que la performance final, el costo y la exactitud del diseño dependen de las mismas.

5.7.3.2 - **Técnicas de diseño y mapeo**

Para producir un array sistólico a partir de la descripción de un algoritmo, el diseñador necesita una comprensión total del mismo, y estar familiarizado con los principios subyacentes a cuatro cosas: el cálculo de tipo sistólico, la aplicación, el algoritmo y la tecnología.

Se han hecho progresos en el desarrollo de técnicas de diseño sistemáticas para automatizar este proceso.

Típicamente, las especificaciones incluyen el tamaño y la topología del array, las operaciones realizadas por cada elemento de procesamiento, el orden y la temporalidad de la comunicación de los datos, y los inputs y outputs.

De forma limitada, estas técnicas pueden tener en cuenta factores tecnológicos y la relación del array sistólico en sí mismo con el resto del sistema. Sin embargo, no son completas ya que solo pueden utilizarse en el nivel de especificaciones y sólo de una manera indirecta.

5.7.3.3 - **Granularidad**

La operación básica realizada en cada ciclo por cada elemento de procesamiento en los diferentes arrays sistólicos varía desde una simple operación de bit, hasta una multiplicación y suma a nivel de palabra, e incluso hasta la completa ejecución de un programa.

La elección del grado de refinamiento está determinado por la aplicación o por la tecnología, o por ambos.

Cuando se programan arrays sistólicos esta granularidad o grado de refinamiento puede también determinarse mediante compromisos entre el grado deseado y el nivel de programabilidad.

5.7.3.4 - **Extensibilidad**

Muchos arrays sistólicos especializados pueden verse como implementaciones hardware de un dado algoritmo.

En tal caso, el procesador sistólico puede ejecutar solamente un determinado algoritmo que está diseñado para un problema de un tamaño específico.

Si uno desea ejecutar el mismo algoritmo para un problema de gran tamaño, o bien debe construir un array mucho mayor o debe particionar el problema.

El primer caso es simple de conceptualizar y solo requiere que más elementos de procesamiento se utilicen para construir una versión extendida del array original.

Sin embargo, en cuanto a implementación se refiere, debemos recordar que pueden existir factores que no afecten la performance en arrays pequeños pero que sí pueden afectarla en sistemas de gran tamaño. Estos factores incluyen la sincronización del reloj, la confiabilidad, las necesidades de alimentación eléctrica, las limitaciones en el tamaño del chip, y las restricciones de los pin de E/S.

5.7.3.5 - **Confiabilidad**

Se pueden utilizar sencillas leyes de la probabilidad para demostrar porqué el incrementar el tamaño del array resulta en la disminución de la confiabilidad a menos que se incorpore redundancia o se disponga de mecanismos tolerantes a fallas.

De hecho, la confiabilidad de un array de procesadores es igual a la confiabilidad de uno de los procesadores elevada a la cantidad de procesadores que contiene el array:

C (conjunto de m procesadores.) = C^m (de 1 procesador)

Debido a que la confiabilidad de un procesador es un número menor a 1, la confiabilidad de todo el array se aproxima rápidamente a cero a medida que el número de procesadores aumenta.

Un buen diseño de estos sistemas tolerantes a fallas tiene como meta la maximización de la confiabilidad mientras que se minimiza el correspondiente overhead.

En los arrays sistólicos las soluciones posibles planteadas incluyen extensiones sencillas de las técnicas bien conocidas utilizadas en las computadoras digitales convencionales. Sin embargo estas técnicas no aprovechan las características propias de los arrays sistólicos ni de los algoritmos que ellos ejecutan.

5.7.3.6 - **Particionamiento de grandes problemas**

Cuando es necesario ejecutar problemas de gran tamaños sin construir arrays sistólicos de gran tamaño, el problema, entonces, debe ser particionado de manera tal que el mismo algoritmo pueda utilizarse para resolver el problema más pequeño, así un array chico y de tamaño fijo puede resolverlo.

Una técnica consiste en identificar particiones del algoritmo y un orden de ejecución de esas particiones de manera tal que la correctitud se mantenga y el array original puede ejecutar cada partición.

El resultado que se observa de esta técnica es que el array "navega" a través del conjunto de los cálculos del algoritmo en un orden correcto hasta "cubrir" todos los cálculos.

Otra técnica trata de replantear el problema a resolver de manera tal que el problema se convierta en un conjunto de problemas más pequeños similares al problema original y que puedan resolverse por el array sistólico.

Mientras que la segunda técnica tiene menos generalidad y es más difícil de automatizar que la primer técnica, puede tener mejor performance cuando se la utiliza.

5.7.4. - **EJEMPLIFICACIÓN**

A continuación veremos algunos ejemplos con algunas soluciones posibles.

5.7.4.1. - **Ejemplo 1)**

Se desea evaluar los polinomios :

$Y_j = \Sigma_i W_i * X_{i+j-1}$; para j de 1 a ∞

o sea :

$Y_1 = W_1 * X_1 + W_2 * X_2 + W_3 * X_3$
$Y_2 = W_1 * X_2 + W_2 * X_3 + W_3 * X_4$
$Y_3 = W_1 * X_3 + W_2 * X_4 + W_3 * X_5$
..... etc .....

Se propone que cada procesador haga la siguiente función :

Fig. 5.13.

siendo $Z_{out} = Z_{in} + W * X_{in}$

Luego el arreglo sistólico sería (para nuestro ejemplo de polinomios de tres componentes) :

Fig. 5.14.

Comenzamos con $Y_0 = 0$ resulta el siguiente proceso :

0 + W1*X1	0 + W2*X1	0 + W3*X1	(1)
0 + W1*X2	W1*X1 + W2*X2	W2*X1 + W3*X2	(2)
0 + W1*X3	W1*X2 + W2*X3	W1*X1 + W2*X2 + W3*X3	(3)
0 + W1*X4	W1*X3 + W2*X4	W1*X2 + W2*X3 + W3*X4	(4)
0 + W1*X5	W1*X4 + W2*X5	W1*X3 + W2*X4 + W3*X5	(5)
....	.....	....	

-(1) Se obtiene W3*X1, se descarta.
-(2) Se obtiene W2*X1 + W3*X2, se descarta.

-(3) Se obtiene W1*X1 + W2*X2 + W3*X3, primer resultado válido.
-(4) Se obtiene W1*X2 + W2*X3 + W3*X4, segundo resultado válido
y así sucesivamente....

5.7.4.2. - **Ejemplo 2) Multiplicación de matrices**

En la Fig. 5.15 se puede ver la estructura de una celda del sistólico.

Aquí las M son celdas (procesadores) multiplicativas-aditivas. En general se requerirán $3(n^2 - n) + 1$ celdas procesadoras, donde n es la dimensión de la matriz, y el cálculo se resolverá en 3(n - 1) períodos (ciclos de reloj).

Veamos paso a paso el caso de la multiplicación de dos matrices de 2 * 2, donde se requieren cinco intervalos para obtener la matriz resultado.

Fig. 5.15.

A = a_{11} a_{12} B = b_{11} b_{12} A * B = C = c_{11} c_{12}
a_{21} a_{22} b_{21} b_{22} c_{21} c_{22}

Intentaremos ver los que sucede en el sistólico por cada instante de tiempo, es decir, mostraremos por cada ciclo como fluye la información dentro del arreglo de celdas procesadoras.

Los PEs en el extremo superior izquierdo y en el inferior derecho no se utilizan en este cálculo, por lo tanto se los inactiva y su función es solamente la de rutear los datos que recibe.

Cantidad de celdas procesadoras : $3(n^2 - n) + 1$

Para n = 2 resulta : 7 celdas procesadoras (son efectivamente 7 las celdas de las cuales se obtienen resultados válidos).

La cantidad de períodos o ciclos en los que se obtiene el resultado final responden a la fórmula : (3n - 1) que resulta en 5 intervalos t.

i) Instante t_1

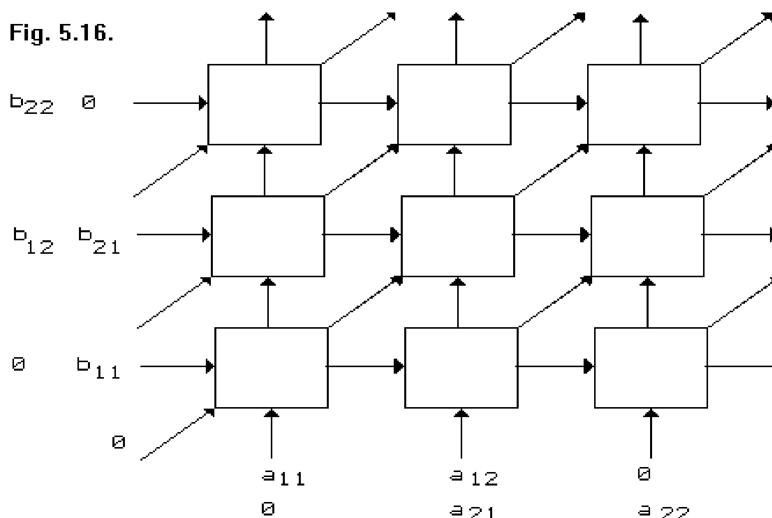


Fig. 5.16.

ii) Instante t_2

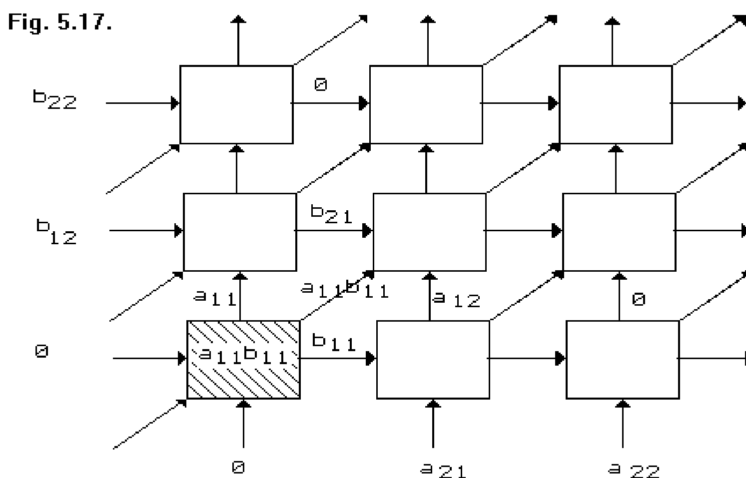
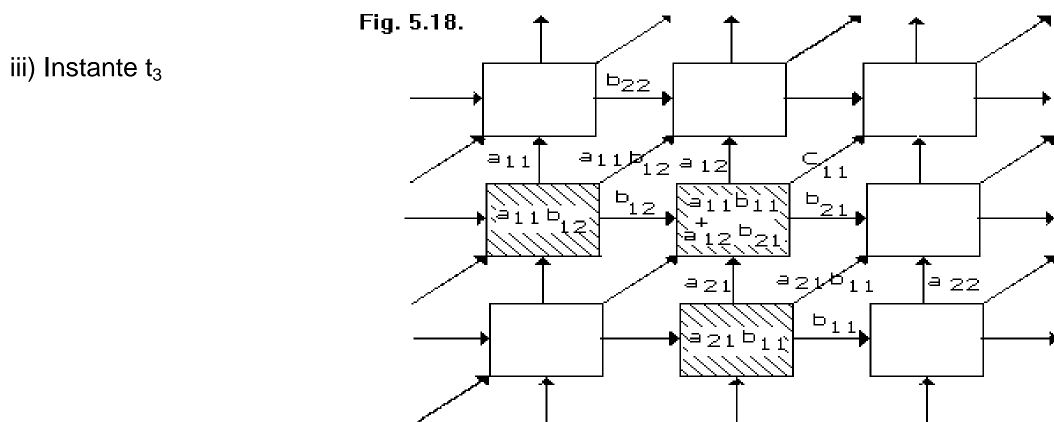
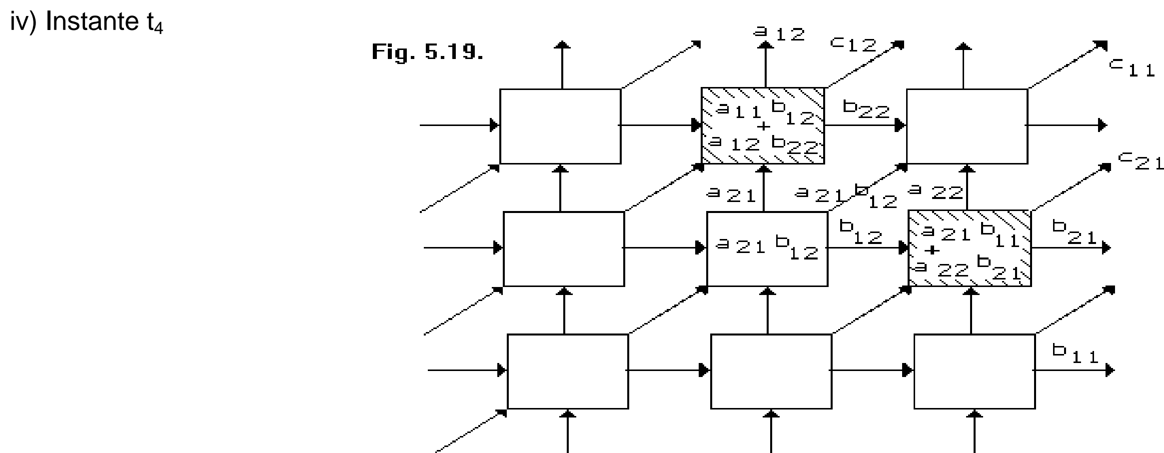


Fig. 5.17.

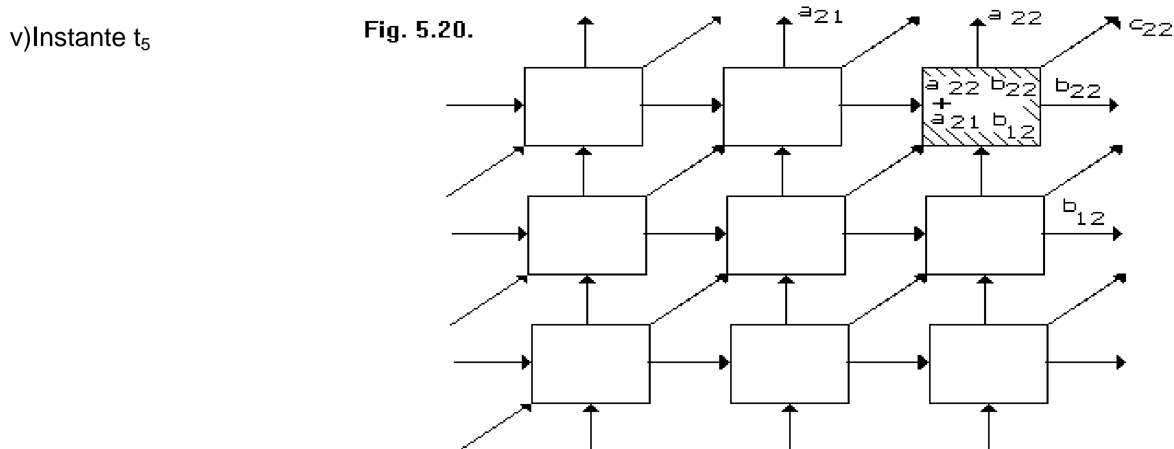
iii) Instante t_3



iv) Instante t_4



v)Instante t_5



La estructura completa para el cálculo de una multiplicación de dos matrices de 3 * 3 responde al siguiente gráfico de la Fig. 5.21.

5.7.5 - **Bloques de construcción universales**

Los arrays sistólicos son más baratos de construir que los otros arrays debido a la profusa repetición de un pequeño número de módulos básicos y sencillos y debido a la alta densidad y eficiencia de sus layouts.

El diseño modular de los arrays sistólicos permite que aquellos diseñadores que desean armar rápidamente prototipos puedan valerse de dispositivos comunes (off-the-shelf), tales como microprocesadores, unidades de aritmética de punto flotante y chips de memoria.

Sin embargo, estas partes no fueron diseñadas para implementaciones de arrays sistólicos y pueden por consiguiente ser inadecuadas para cumplir los requerimientos del diseño.

Esto desembocó en el desarrollo de "bloques universales de construcción" que son chips que pueden utilizarse para muchos arrays sistólicos.

Algunos chips comercialmente disponibles de este tipo que merezcan esta definición incluyen al Transputer INMOS, al TI TMS32010 y TMS32020, el chip de dataflow NEC PD 7281, etc.

Los problemas que implica el uso de bloques de construcción programables incluyen el desarrollo de herramientas de programación de ayuda a los diseñadores y la provisión de un soporte para interconexiones flexibles.

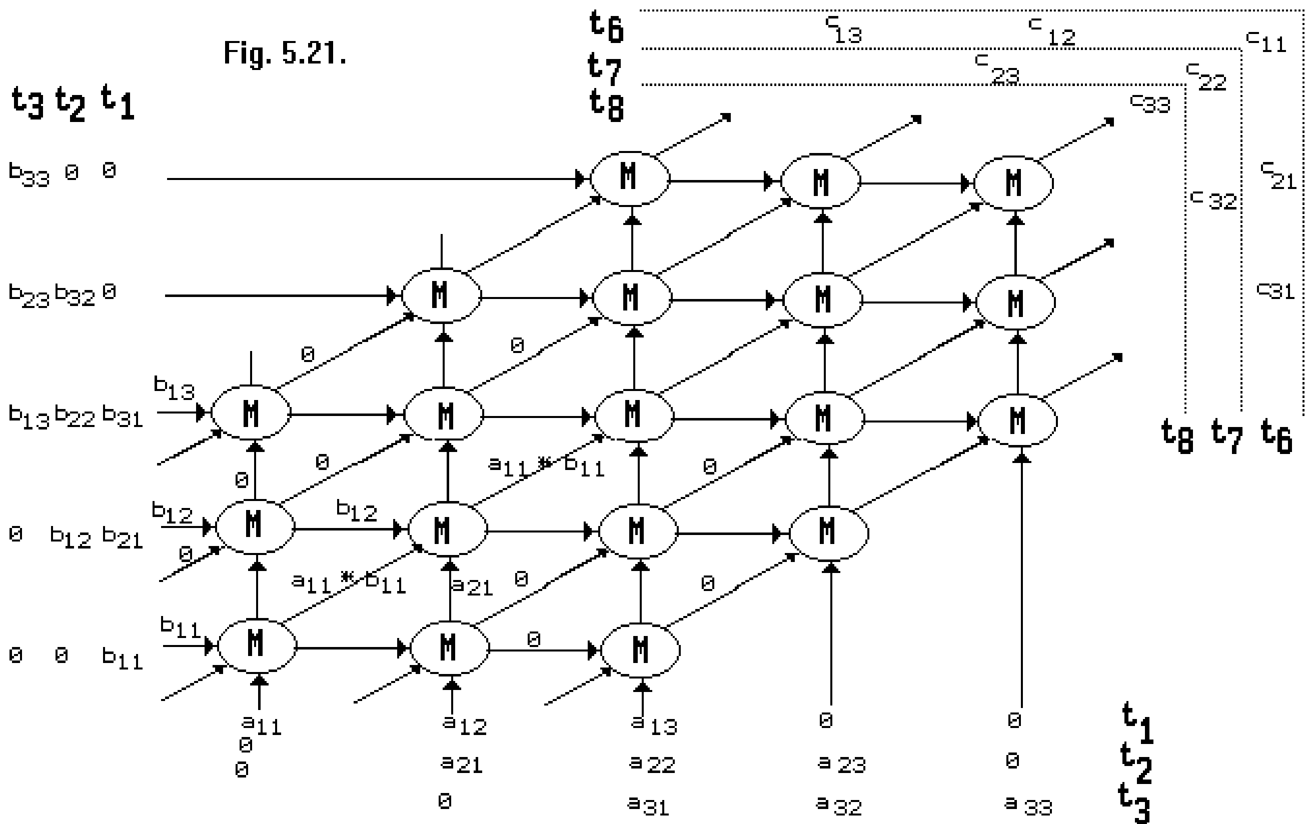


Fig. 5.21.

5.7.6 - **Integración en sistemas existentes**

A pesar de que los arrays sistólicos proveen una alta performance, su integración dentro de sistemas ya existentes es no trivial debido al amplio bandwidth de E/S que se produce, sobretodo cuando el problema debe ser particionado y los datos de entrada deben ser accedidos repetidamente.

Los problemas adicionales que deben resolverse en aquellos sistemas que cuentan con gran cantidad de arrays sistólicos incluyen la interconexión con el host, el subsistema de memoria que soporte los array sistólicos, el bufferizado y el acceso a datos para obtener la distribución especial de datos de E/S, y la multiplexación y desmultiplexación de datos cuando son insuficientes las puertas de E/S.

EJERCICIOS

1) Una arquitectura de procesador array es sincrónica o asincrónica ? Justifique.
2) Cuál es la diferencia entre las dos implementaciones más clásicas de procesadores array ?
3) Un PE es una CPU ? Justifique.
4) Qué es y para qué se utiliza el esquema de enmascaramiento de PEs ?
5) El enmascaramiento y la función de ruteo en arquitecturas SIMD son la misma cosa ? Justifique.
6) Construya el vector de máscaras y la función de ruteo para cada uno de los pasos del ejemplo del apunte (Fig. 5.6).
7) Cómo trabaja un procesador array de tipo bit-plane ?
8) Construya una función AND para un computador de memoria asociativa que opere solamente sobre las filas pares de la memoria.
9) Qué es un array sistólico ? Cómo funciona ?
8) Puede clasificarse a los sistólicos como arquitecturas MISD ? Discuta.
10) Puede un array sistólico ser un procesador de propósito general ? Justifique.
11) Qué mecanismo se utiliza para manejar grandes problemas que se desea procesar en arrays sistólicos ?
12) Los procesadores sistólicos son de gran utilidad para aquellos problemas de tipo compute-bound ? Justifique.
13) Sea el esquema de interconexión de celdas en un array sistólico de la Figura 5.22.
Cómo deben ingresar los valores de las matrices A y B de 2 x 2 para evaluar la matriz C = A * B ? Indique expresamente paso a paso la evaluación no olvidando los valores de sincronismo.

Ayuda : los caminos horizontales y verticales son solo para datos, los caminos en diagonal se utilizan para rutear los resultados de las multiplicaciones a otras celdas que realizan las sumas.

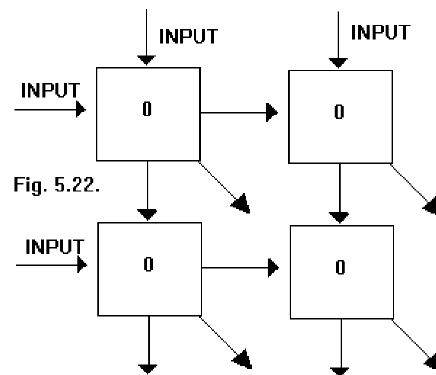


Fig. 5.22.

CAPITULO 6

MULTIPROCESADORES (MIMD)

6.1 - Generalidades de Multiprocesadores.

Pueden clasificarse en esta categoría muchos sistemas multiprocesadores y sistemas multicomputadores.

Un multiprocesador se define como una computadora que contiene dos o más unidades de procesamiento que trabajan sobre una memoria común bajo un control integrado.

Si el sistema de multiprocesamiento posee procesadores de aproximadamente igual capacidad, estamos en presencia de multiprocesamiento simétrico; en el otro caso hablamos de multiprocesamiento asimétrico.

Todos los procesadores deben poder acceder y usar la memoria principal. De acuerdo a esta definición se requiere que la memoria principal sea común y solamente existen pequeñas memorias locales en cada procesador.

Si cada procesador posee una gran memoria local se lo puede considerar un sistema de multicomputadoras, el cual puede ser centralizado o distribuido.

Todos los procesadores comparten el acceso a canales de E/S, unidades de control y dispositivos.

Para el sistema de multiprocesamiento debe existir un sistema operativo integrado, el cual controla el hardware y el software y debe asegurar la interacción entre los procesadores y sus programas al nivel elemental de dato, conjunto de datos y trabajos.

Una computadora MIMD intrínseca implica interacciones entre n procesadores debido a que todos los flujos de memoria se derivan del mismo espacio de datos compartido por todos los procesadores. Si los n flujos de datos provienen de subespacios disjuntos de memorias compartidas, entonces estamos en presencia del denominado operación SISD múltiple, que no es otra cosa que un conjunto de n monoprocesadores SISD.

Una MIMD intrínseca está fuertemente acoplada si el grado de interacción entre los procesadores es alto. De otra manera consideramos el sistema como débilmente acoplado. Muchos sistemas comerciales son débilmente acoplados, a saber, la IBM 370/168, Univac 1100/80, IBM 3081/3084, etc.

6.2 - MULTIPROCESADORES Y MULTICOMPUTADORES

Existen similitudes entre los sistemas multiprocesadores y multicomputadores debido a que ambos fueron pensados con un mismo objetivo: dar soporte a operaciones concurrentes en el sistema. Sin embargo, existen diferencias importantes basadas en el alcance de los recursos compartidos y la cooperación en la solución de un problema.

Un sistema multicomputador consiste de diversas computadoras autónomas que pueden o no comunicarse entre sí.

Un sistema multiprocesador está controlado por un sistema operativo que provee la interacción entre los procesadores y sus programas a nivel de dato, proceso y archivo.

6.3 - FORMAS DE ACOPLAMIENTO

Existen dos modelos arquitectónicos diferentes para los sistemas multiprocesadores: Fuertemente acoplado y Débilmente acoplado. Los sistemas fuertemente acoplados se comunican a través de una memoria común. De allí que el promedio de velocidad con la cual un procesador puede comunicarse con otro es del orden del bandwidth de la memoria.

Puede existir una pequeña memoria local o un buffer de alta velocidad (cache) en cada procesador.

Existe una completa conectividad entre los procesadores y la memoria. Esta conectividad puede alcanzarse insertando una red de interconexión entre los procesadores y la memoria; o mediante una memoria multipuertas.

Uno de los factores que limitan el crecimiento de los sistemas fuertemente acoplados es la degradación debido a la contención de memoria que ocurre cuando dos o más procesadores intentan acceder la misma unidad de memoria concurrentemente.

Puede reducirse el grado de conflictividad incrementando el grado de interleaving. Sin embargo, esto debe acompañarse de una cuidadosa asignación de los datos a los módulos de memoria.

Los sistemas multiprocesadores débilmente acoplados no tienen, en general, el grado de conflictos sobre la memoria de los fuertemente acoplados.

En este sistema cada procesador tiene un conjunto de dispositivos de E/S y una gran memoria local a donde accede para obtener la mayoría de sus datos e instrucciones.

Nos referiremos al procesador, sus dispositivos de E/S y su memoria local como al módulo computador.

Los procesos que se ejecutan en diferentes procesadores se comunican intercambiando mensajes a través de un sistema de transferencia de mensajes. El grado de acoplamiento en tales sistemas es realmente muy débil, de allí que se los conozca también como sistemas distribuidos.

El factor determinante del grado de acoplamiento es la topología de la comunicación del sistema de transferencia de mensajes asociado.

Los sistemas débilmente acoplados son eficientes cuando la interacción entre las tareas es mínima.

Los sistemas fuertemente acoplados pueden soportar una gran interacción entre las tareas sin un deterioro significativo de la performance.

La Fig. 6.1. muestra un ejemplo de un módulo computador de un multiprocesador débilmente acoplado no

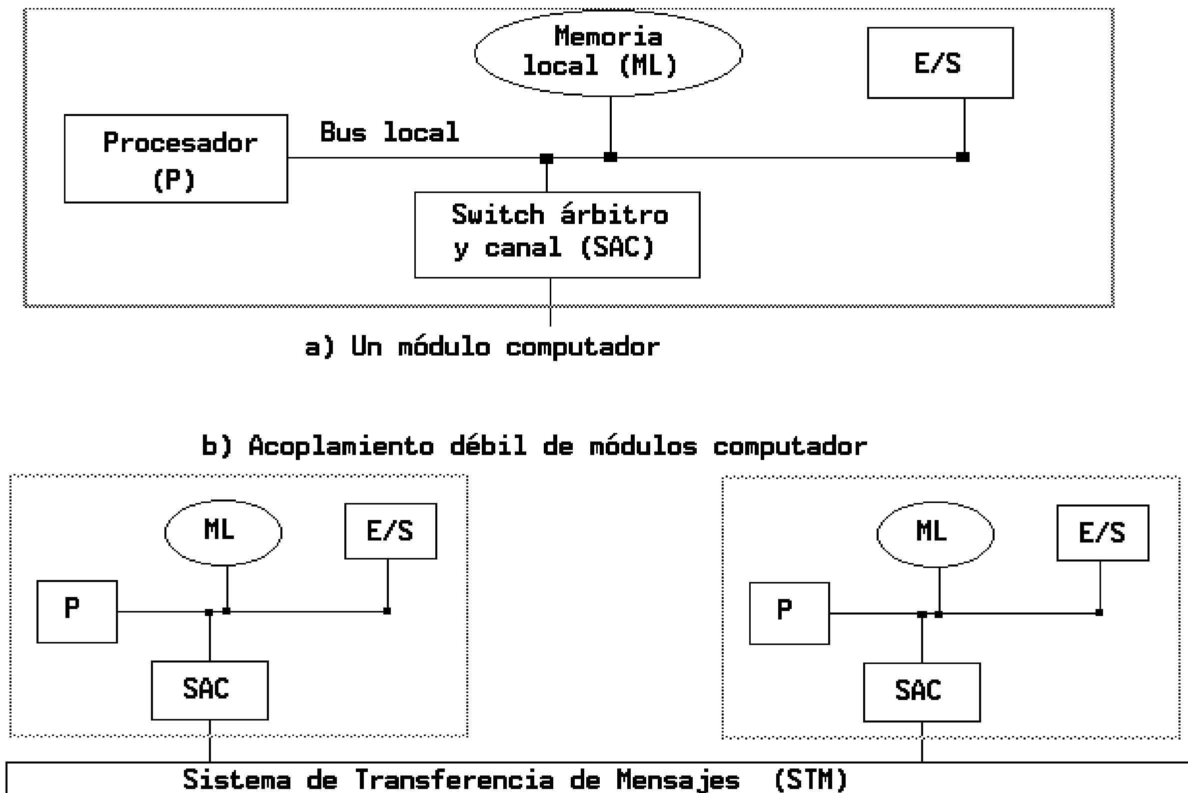


Fig. 6.1. - Sistema multiprocesador no jerárquico débilmente acoplado.

jerárquico. Consiste en un procesador, una memoria local, dispositivos de E/S locales y una interfase a otros módulos computadores.

La interfase puede contener un switch árbitro y un canal. La Fig. 6.1. muestra también la conexión entre los módulos computador y el sistema de transferencia de mensajes (STM).

Si los pedidos para dos o más módulos computador colisionan al acceder un segmento físico del STM, el árbitro es el responsable de elegir uno de los pedidos simultáneos de acuerdo a una determinada disciplina de servicio.

Es también responsable de hacer esperar los otros pedidos hasta que se complete la atención del pedido actual.

El canal que se encuentra dentro del SAC puede tener una memoria de comunicación de alta velocidad a efectos de bufferizar los bloques de transferencia de mensajes. La memoria de comunicación es accesible por todos los procesadores.

En la Fig. 6.2 podemos ver un esquema de multiprocesador fuertemente acoplado.

Este consiste de P procesadores, S módulos de memoria y D canales de Entrada/Salida. Estas unidades están conectadas mediante tres redes de interconexión, a saber, la red de interconexión entre los procesadores y los IOP, y la red de interconexión de interrupciones-señales.

Los conflictos de acceso a memoria por varios procesadores son resueltos por la red de interconexión procesador-memoria. Para evitar excesivos conflictos, la cantidad de módulos de memoria S es generalmente tan grande como P.

Otro método para reducir el grado de conflictos es asociar un área de almacenamiento reservada para cada procesador. Esta es la memoria local no mapeada que se usa para almacenar código Kernel y tablas del sistema operativo muy utilizadas por los procesos que se ejecutan en tal procesador.

Se puede agregar también a esta configuración una memoria cache propia de cada procesador a fin de disminuir las referencias a memoria principal.

La red de interconexión de interrupciones-señales permite que cada procesador envíe directamente una interrupción a otro procesador. La sincronización entre procesos se ve facilitada por esta red. Esta red puede actuar como un procesador de fallas ya que puede enviar una alarma originada por hardware a los procesadores que sí funcionen.

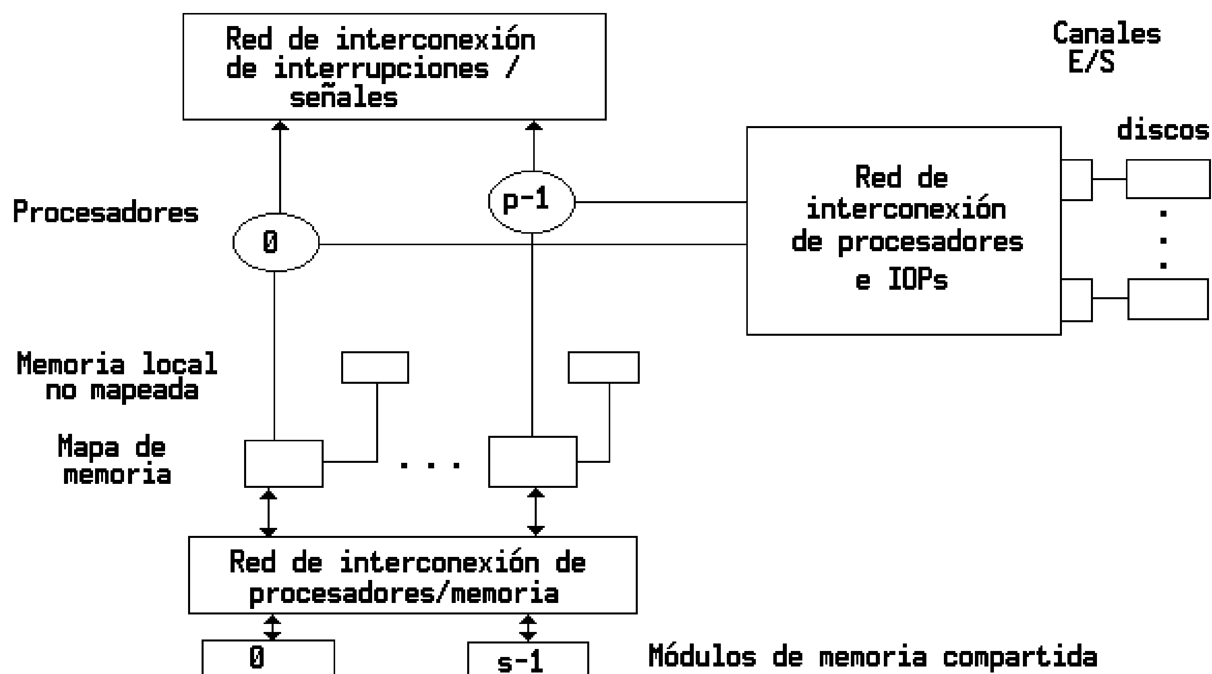


Fig. 6.2. - Configuración multiprocesador fuertemente acoplado.

El conjunto de procesadores puede ser homogéneo o heterogéneo. Es homogéneo si los procesadores son funcionalmente idénticos.

Pero aún siendo homogéneos pueden ser simétricos o asimétricos dependiendo de que dos unidades funcionalmente idénticas difieran en cuanto a dimensiones tales como accesibilidad de E/S, performance o confiabilidad. Esta última configuración de la Fig. 6.2 como procesador fuertemente acoplado es también conocida como sistema de multiprocesadores *diádicos.*

6.4. - DEFINICION DE ARQUITECTURAS PARALELAS

Problemas. Se han propuesto diversas definiciones para arquitecturas paralelas. La dificultad en definir con precisión el término está entrelazada con el problema de especificar una taxonomía de arquitecturas paralelas. El problema central para poder especificar una definición y consiguientemente la taxonomía para las modernas arquitecturas paralelas es lograr satisfacer el siguiente conjunto de imperativos :

*) Excluir las arquitecturas que incorporan solamente mecanismos de paralelismo de bajo nivel y que se han popularizado tanto como característica típica de las modernas computadoras.

*) Mantener los elementos útiles de la clasificación de Flynn tales como los flujos de datos e instrucciones.

*) Incluir los procesadores vectoriales pipelinizados y otras arquitecturas que intuitivamente ameritan incluirse como arquitecturas paralelas, pero que no se ajustan fácilmente al esquema de Flynn.

Examinaremos cada una de estos imperativos así como obtendremos una definición que los satisface totalmente y provee una base para una razonable taxonomía.

6.4.1. - Paralelismo de bajo nivel

Existen dos razones para excluir las máquinas que utilizan mecanismos de paralelismo de bajo nivel del conjunto de arquitecturas paralelas.

Primero, si no adoptamos un standard riguroso prácticamente la mayoría de las computadoras modernas serían "arquitecturas paralelas", anulando la utilidad del término en sí. Y segundo, las arquitecturas que solamente tienen las características que vamos a enunciar a renglón seguido no ofrecen un marco de referencia explícito y coherente para desarrollar soluciones paralelas de alto nivel :

Pipelining de instrucciones la descomposición de la ejecución de una instrucción en una serie lineal de etapas autónomas, permitiendo que cada etapa simultáneamente realice una porción del procesamiento de la instrucción (por. ej. decodificación, cálculo de la dirección efectiva, levantar operandos de memoria, ejecutar y almacenar).

Múltiples unidades funcionales en la CPU proveyendo unidades funcionales independientes para la ejecución concurrente de operaciones aritméticas y booleanas.

Procesadores separados para E/S y CPU liberando a la CPU del control sobre las entradas/salidas mediante el uso de procesadores dedicados, solución que abarca desde los controladores más sencillos de E/S hasta las complejas unidades de procesamiento periféricas.

A pesar de que estas características contribuyen significativamente a la performance, su presencia no hace que una computadora posea una arquitectura paralela.

6.4.2. - **Taxonomía de Flynn**

La taxonomía de Flynn clasifica las arquitecturas de las computadoras según la presencia de únicos o múltiples flujos de datos e instrucciones. Hemos visto ya en el capítulo 4 las cuatro categorías de esta clasificación.

SISD define las computadoras seriales.

MISD implica que muchos procesadores aplican diferentes instrucciones al mismo dato, esta posibilidad hipotética se considera generalmente impracticable.

SIMD implica que múltiples procesadores ejecutan simultáneamente la misma instrucción sobre diferentes datos.

MIMD implica que múltiples procesadores ejecutan autónomamente diversas instrucciones sobre diversos datos.

Si bien estas distinciones proveen elementos útiles para caracterizar arquitecturas, no bastan para clasificar varias de las computadoras modernas.

Por ejemplo, los procesadores vectoriales pipelinizados merecen ser incluidos como arquitecturas paralelas, ya que muestran una concurrencia substancial en la ejecución aritmética y pueden manejar cientos de elementos de vectores en forma paralela, no obstante lo cual no se ajustan a los parámetros de la clasificación de Flynn, debido a que, si los consideramos SIMD carecen de procesadores que ejecutan la misma instrucción en pasos bien acotados, y si por otra parte los clasificaremos como MIMD les falta la autonomía asincrónica de la categoría.

6.4.3. - **Definición y taxonomía**

Un primer paso para proveer una taxonomía satisfactoria es articular una definición de arquitecturas paralelas. Esta definición debe incluir las computadoras que la clasificación de Flynn no puede manejar y debe excluir aquellas que incorporan el paralelismo de bajo nivel.

Por lo tanto, una arquitectura paralela provee un explícito marco de referencia de alto nivel para el desarrollo de soluciones de programación paralelas logrado mediante múltiples procesadores (simples o complejos) que cooperan para resolver problemas a través de ejecución concurrente.

La Fig. 6.3 muestra una taxonomía basada en las imperativas discutidas anteriormente y la definición propuesta.

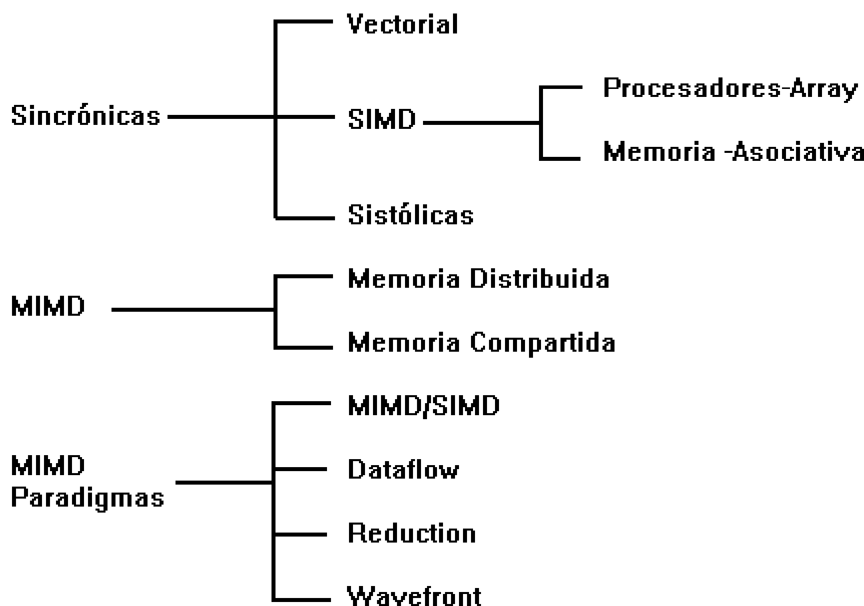


Fig. 6.3. - Una taxonomía de alto nivel para arquitecturas de computadores paralelos.

Esta taxonomía informal utiliza categorías de alto nivel para delinear los principales planteamientos sobre las arquitecturas de las computadoras paralelas y para mostrar que estos planteamientos definen un espectro coherente de alternativas arquitecturales. Las definiciones de cada categoría las definimos a continuación.

Esta taxonomía no intenta suplantar a aquellas construidas con un mayor esfuerzo formal. Tales taxonomías proveen subcategorías que reflejan alteraciones de características arquitecturales y cubren las características de bajo nivel.

6.5. - **ARQUITECTURAS SINCRÓNICAS**

Las arquitecturas paralelas sincrónicas coordinan operaciones concurrentes en pasos acotados mediante relojes globales, unidades centrales de control o controladores de la unidad vectorial.

6.5.1. - **Procesadores Vectoriales Pipelinizados**

Los procesadores vectoriales se caracterizan por múltiples unidades funcionales pipelinizadas que operan concurrentemente e implementan operaciones aritméticas y booleanas tanto escalares como matriciales.

Debido a que tales arquitecturas proveen paralelismo a nivel de tareas, puede argumentarse con ciertas reservas que son MIMD, aún cuando las capacidades de procesamiento vectorial son el aspecto fundamental de sus diseños.

6.6. - **Arquitecturas SIMD y de Procesador Array**

Las arquitecturas SIMD y las de los procesadores array fueron ya vistas con mayor detalle en el capítulo 5.

6.7. - **ARQUITECTURAS MIMD**

Hemos ya discutido en este capítulo las características básicas de las arquitecturas MIMD. En el siguiente capítulo veremos los casos particulares según que la memoria sea compartida o distribuida, así como algunas de las diferentes topologías que se utilizan.

6.8. - **PARADIGMAS BASADOS EN ARQUITECTURAS MIMD.**

Las arquitecturas híbridas MIMD/SIMD, las máquinas de Reducción, las arquitecturas Dataflow y los Wavefront array son arquitecturas igualmente difíciles de acomodar ordenadamente en una clasificación de las arquitecturas paralelas.

Cada una de estas arquitecturas está basada en los principios de operación asincrónica y manejo concurrente de múltiples flujos de datos e instrucciones.

Sin embargo cada una de ellas se basa, a su vez, en algún principio muy distintivo que se suma a las características propias de ser MIMD y que, por lo tanto, merecen un tratamiento especial.

Por esta razón es que incluiremos estas arquitecturas en el capítulo 9 como "Arquitecturas Nuevas" para destacar, justamente, sus características particulares.

EJERCICIOS

1) Qué es una arquitectura MIMD y cómo funciona ? Grafíquelo.
2) Cuál es la diferencia entre un MIMD fuertemente acoplado y un MIMD débilmente acoplado ?
3) Falso o Verdadero : No mejora la performance de un sistema multiprocesador débilmente acoplado el contar con memoria interleaved. Justifique.
4) En qué caso son eficientes los sistemas multiprocesadores fuertemente acoplados ?
5) Qué término se utiliza para indicar cuando un sistema multiprocesador posee procesadores de aproximadamente igual capacidad ?
6) A qué se denomina paralelismo de bajo nivel y en qué situaciones es factible encontrarlo ?
7) Indique cuáles de los elementos que se enumeran constituyen formas de paralelismo de bajo nivel :

- pipelining de instrucciones
- procesadores diádicos
- pipelines aritméticos
- IOP's
- pipeline vectorial

8) Cuál es la diferencia entre una arquitectura sincrónica y una no sincrónica ?
9) En base a qué se pueden clasificar las arquitecturas ?

CAPITULO 7

ARQUITECTURAS DISTRIBUIDAS

7.1 - MULTIPROCESADORES (MIMD)

Un multiprocesador se define como una computadora que contiene dos o más unidades de procesamiento que trabajan sobre una memoria común bajo un control integrado (recordemos su arquitectura según la clasificación de Flynn).

Si el sistema de multiprocesamiento posee procesadores de aproximadamente igual capacidad, estamos en presencia de multiprocesamiento simétrico, en el otro caso hablamos de multiprocesamiento asimétrico.

Todos los procesadores deben poder acceder y usar la memoria principal. De acuerdo a esta definición se requiere que la memoria principal sea común y solamente existen pequeñas memorias locales a cada procesador.

Si cada procesador posee una gran memoria local se lo puede considerar un sistema de multicomputadoras, el cual puede ser centralizado o distribuido.

Todos los procesadores comparten el acceso a canales de E/S, unidades de control y dispositivos.

Para el sistema de multiprocesamiento debe existir un sistema operativo integrado el cual controla el hardware y el software y debe asegurar la interacción entre los procesadores y sus programas a un nivel elemental de dato, conjunto de datos, tareas y trabajos.

Aún cuando la ejecución de los procesos en arquitecturas MIMD se sincroniza mediante el pasaje de mensajes por medio de una red de interconexión o accediendo a datos en unidades de memoria compartida, las arquitecturas MIMD son computadoras asincrónicas caracterizadas por un hardware de control descentralizado.

La efectividad del costo de n Sistemas Procesadores respecto de n procesadores aislados alienta la experimentación en las MIMD.

7.2 - SISTEMAS DISTRIBUIDOS

Hemos dicho ya que los sistemas MIMD débilmente acoplados reciben también el nombre de Sistemas Distribuidos. Pasamos ahora a tratar tales sistemas.

Existen cuatro grandes razones para construir Sistemas Distribuidos, a saber:

- el compartir los recursos,
- la velocidad del cómputo,
- la confiabilidad y
- la comunicación.

7.2.1 - Compartir Recursos

Si una cantidad de nodos (sitios, computadores) están conectados a otros, entonces un usuario en un nodo puede hacer uso de los recursos disponibles en el otro.

Por ejemplo, un usuario en el nodo A desea utilizar una impresora láser que se encuentra en el nodo B, mientras tanto el usuario en el nodo B quiere acceder a un archivo que está en el nodo A.

En general, en un sistema distribuido están provistos los mecanismos para compartir archivos en sitios remotos, o para procesar información en una base de datos distribuida, e imprimir archivos en sitios remotos, utilizar en forma remota dispositivos hardware especializados (como por ejemplo array processor), y otras operaciones.

7.2.2 - Velocidad de Cómputo

Si un cálculo particular puede dividirse en una cierta cantidad de subcómputos que puedan ejecutarse concurrentemente, entonces la disponibilidad de un sistema distribuido nos permite justamente distribuir el cálculo entre varios nodos.

Además si un nodo se encuentra sobrecargado de trabajos puede rutearlos hacia otro nodo menos cargado. Este movimiento de trabajos se denomina "compartir la carga".

7.2.3 - Confiabilidad

Si un nodo falla en un sistema distribuido, los sitios restantes pueden potencialmente continuar la operatoria.

Si el sistema está compuesto por una gran cantidad de instalaciones autónomas (por. ej. computadoras de propósito general) entonces la falla de una de ellas no afectaría el resto.

Si por otra parte el sistema está compuesto de máquinas pequeñas, cada una de las cuales es responsable de alguna función crucial del sistema, entonces una sola falla puede efectivamente detener la operatoria total del sistema.

En general, si existe la suficiente redundancia en el sistema (tanto de hardware como de software) luego el sistema puede continuar operando aún cuando algunos de sus nodos fallen.

7.2.4 - **Comunicación**

Cuando una cantidad de nodos están conectados a otros mediante una red de comunicación, los usuarios en diferentes nodos tienen la oportunidad de intercambiar información. En un sistema distribuido nos referiremos a esta actividad como "correo electrónico".

Cada usuario de la red tiene asociado una única dirección (en el sentido de domicilio -mailbox-) y puede intercambiar correspondencia con otro usuario en el mismo nodo o en nodos diferentes.

Este correo no es interpretado por el sistema operativo.

7.3 - **TOPOLOGIA DE LA RED**

Los nodos en un sistema pueden estar comunicados de diferentes maneras. Cada configuración tiene sus ventajas y desventajas.

Veremos algunas de las configuraciones que han sido implementadas hasta la fecha y las compararemos respecto de los siguientes criterios :

- **Costo básico** : Cuánto cuesta unir los diferentes nodos en el sistema ? Cuánto cuesta anexar un nodo ?
- **Costo de comunicación** : Cuánto tiempo tarda entregar un mensaje del nodo A al nodo B ?
- **Confiabilidad** : Si una conexión a un nodo falla, pueden comunicarse los otros nodos entre sí ?

7.4. - **ARQUITECTURAS DE MEMORIA DISTRIBUIDA**

Las arquitecturas de memorias distribuidas conectan nodos de procesamiento consistentes de un procesador autónomo y su memoria local (ver Fig. 7.1) con una red de interconexión de procesador-a-procesador.

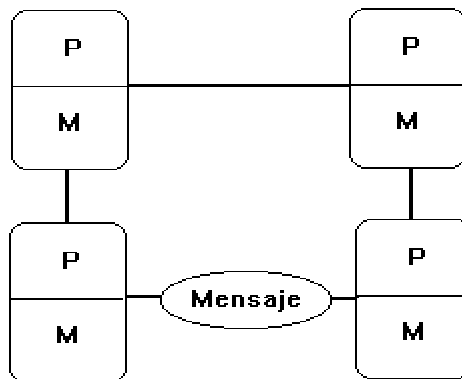


Fig. 7.1. - Estructura MIMD con memoria distribuida.

Los nodos comparten datos pasándose mensajes explícitamente a través de la red, debido a que no existe una memoria compartida.

Como un producto de las investigaciones de los años 80 estas arquitecturas se construyeron en un esfuerzo de proveer una arquitectura multiprocesador que pudiera "expandirse" (crecer en cantidad de procesadores) y satisficiera los requerimientos de performance de voluminosas aplicaciones científicas caracterizadas por referencias locales a datos.

Se han propuesto varias topologías de red de interconexión que permiten esta expansibilidad arquitectural y una mayor performance para programas paralelos y que difieren en los patrones de comunicación entre los procesadores. Veremos algunas a continuación.

7.4.1 - **Totalmente conectada (Completely connected)**

En esta topología cualquier nodo en el sistema está conectado a todos los otros nodos de la red. (Fig. 7.2).

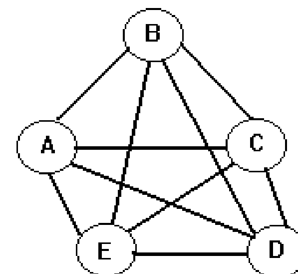


Fig. 7.2. - Totalmente conectada.

El costo básico de esta configuración es alto, ya que debe existir una conexión directa entre cada dos nodos. El costo de anexar un nuevo nodo crece según X, donde X es la cantidad de nodos que contiene la red.

Sin embargo, en este entorno los mensajes entre nodos pueden entregarse muy velozmente, cualquier mensaje utiliza solamente una conexión para viajar entre dos nodos.

Estos sistemas son muy confiables ya que deben fallar muchos nodos para que el sistema completo falle.

Un sistema está particionado si se ha dividido en dos subsistemas que no pueden comunicarse entre sí.

7.4.2 - **Parcialmente conectada (Partially connected)**

En una red parcialmente conectada existen conexiones directas entre dos nodos pero no para todos los nodos (Fig. 7.3). De allí que el costo básico de la red sea menor.

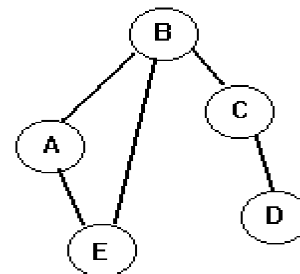


Fig. 7.3. - Parcialmente conectada.

Un mensaje de un nodo a otro puede requerir viajar entre varios nodos antes de arribar a su destino, lo que resulta en una comunicación más lenta.

No es tan confiable como la totalmente conectada.

Por ejemplo en la figura si falla la conexión entre el nodo B y el nodo C entonces la red está particionada en dos subsistemas. Para minimizar esta posibilidad en general cada nodo está conectado con por lo menos otros dos nodos.

7.4.3 - **Jerárquica o Arbol (Tree)**

En una red jerárquica los nodos están organizados como un árbol (Fig. 7.4). Es una organización común en las redes de computadoras corporativas donde las oficinas individuales están conectadas a una oficina central local, las que a su vez cuelgan de las oficinas regionales y esta finalmente de las oficinas en la central.

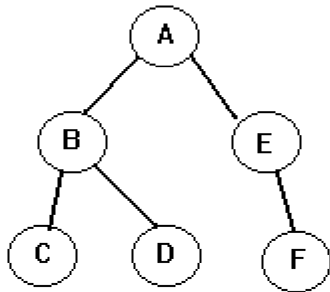


Fig. 7.4. - Jerárquica o Arbol.

Cada nodo (excepto la raíz) tiene un único antecesor y varios hijos. El costo básico de esta configuración es menor que el de la parcialmente conectada.

Un padre y su hijo se comunican directamente. Los hermanos se comunican entre sí solo a través del padre. En forma similar los primos solo pueden comunicarse a través de sus abuelos.

Si un nodo falla todos sus hijos quedan incomunicados con el resto de la red. En general, la falla de cualquier nodo puede particionar la red en varios subárboles disjuntos.

Aún cuando se han propuesto varias topologías basadas en la estructura de árbol, los árboles binarios completos han sido la variante más analizada.

Se han empleado varias estrategias para reducir el diámetro de comunicación de estas topologías (2(n-1) para un árbol binario completo de n niveles y 2^n-1 procesadores). Algunas soluciones incluyen agregar conexiones extra a la red para unir todos los nodos de un mismo nivel.

7.4.4 - **Estrella (Star)**

En una red estrella uno de los nodos está conectado a todos los demás (Fig. 7.5).

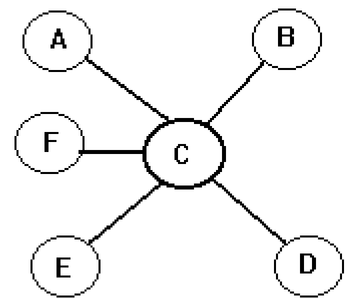


Fig. 7.5. - Estrella

El costo básico de este sistema es lineal respecto del número de nodos.

El costo de comunicación es también bajo, ya que un mensaje del proceso A al proceso B solamente requiere de dos transferencias (Desde A a C y de C a B). Sin embargo esta velocidad puede no ser tal ya que el nodo central puede convertirse en un cuello de botella.

Mientras haya pocos mensajes la velocidad se mantendrá alta. En algunos sistemas estrella el nodo central está totalmente dedicado al pasaje de mensajes.

Si el nodo central falla la red queda totalmente particionada.

7.4.5 - **Anillo (Ring)**

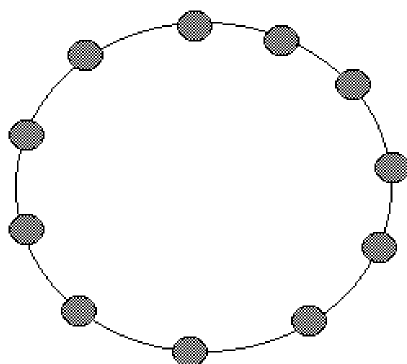


Fig. 7.6. - Anillo

En una red anillo cada nodo está conectado a solo otros dos nodos (Fig. 7.6).

El anillo puede ser unidireccional o bidireccional. En una arquitectura unidireccional un nodo puede transmitir información hacia uno solo de sus vecinos. Todos los nodos entregan información hacia la misma dirección.

Esta topología ha sido muy utilizada por IBM para sus redes denominadas "Token Ring".

En una arquitectura bidireccional un nodo puede transmitir información hacia cualquiera de sus vecinos.

De forma típica, se utilizan paquetes de mensajes de tamaño fijo incluyendo un campo que indica el nodo destino.

El costo básico de un anillo es nuevamente lineal respecto de la cantidad de nodos.

Sin embargo el costo de comunicación puede ser bastante alto ya que un mensaje de un nodo a otro debe viajar alrededor del anillo hasta que llega a destino. En un anillo unidireccional deberá recorrer a lo sumo n-1 nodos, en tanto que en un bidireccional a lo sumo n/2.

En una red bidireccional deben fallar dos conexiones para que la red se particione. En un anillo unidireccional la falla de un solo nodo particionará la red.

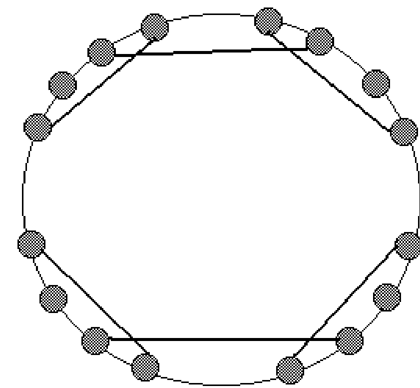


Fig. 7.7. - Anillo cordado.

Un remedio que suele utilizarse es el proveer al anillo de una doble conexión como puede verse en la Fig. 7.7. Esta arquitectura se denomina *anillo cordado (chordal ring)*.

Las topologías de anillo son más apropiadas para un pequeño número de procesadores que ejecutan algoritmos en donde lo predominante no debe ser la comunicación de datos.

En el Capítulo 20 de Sistemas Distribuidos, veremos algo más respecto de las topologías de Anillo.

7.4.6 - **Red con vecinos cercanos (Mesh connected)**

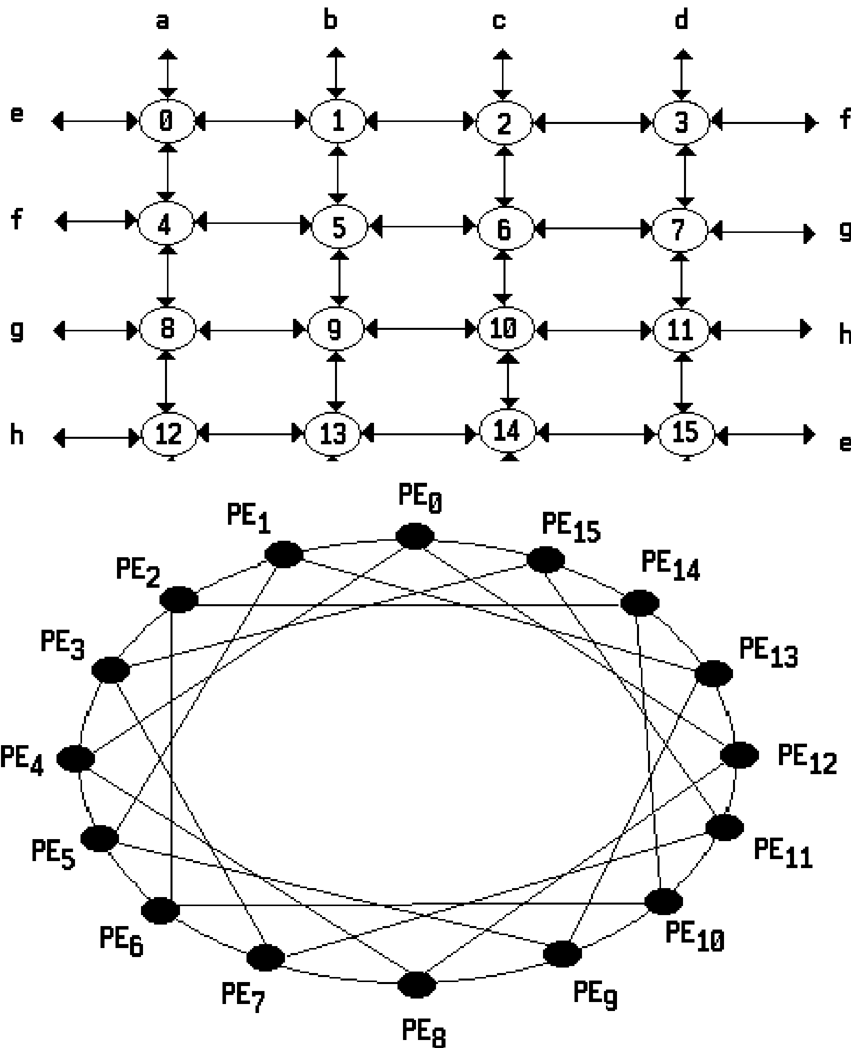


Fig. 7.9. - Red con vecinos cercanos vista de otra forma.

Para simplificar la explicación de este tipo de red también conocida como **Near-neighbor mesh** (mesh: grilla, malla) la aplicaremos a un ejemplo específico.

Sean 16 nodos N numerados del 0 al 15. Cada N(i) puede enviar mensajes a cada N(**i+1**), N(**i-1**), N(**i-r**) y N(**i+r**); siendo r la raíz cuadrada de N (generalmente se elige N como un numero de cuadrado perfecto).

En la Fig. 7.8 puede verse esta estructura y en la Fig. 7.9 puede verse la misma estructura dibujada de otra forma.

Cada N(i) está conectado aquí a sus cuatro vecinos más cercanos en la red. Este tipo de red es una red parcialmente conectada.

Los pasos que insume transferir datos desde un nodo N(i) a un nodo N(j) en una red de tamaño N es un valor cuya cota superior es :

I = SQRT(N) - 1

En una red de por ejemplo 64 nodos se necesitan a lo sumo 7 pasos para rutear un dato de cualquier nodo a otro.

Se puede aumentar la comunicación agregando conexiones adicionales de tipo diagonal o usando buses para conectar nodos por fila o columna.

La correspondencia entre estas topologías y los algoritmos orientados a matrices alientan las investigaciones sobre estas arquitecturas.

7.4.7 - **Cubo**

En la Fig. 7.10 se ilustra un cubo tridimensional.

Las líneas verticales conectan vértices cuyas direcciones difieren en el bit más significativo. Los vértices en ambos extremos de las líneas diagonales difieren en el bit de posición media. Las líneas horizontales difieren en el bit menos significativo.

Este concepto de cubo puede extenderse a un espacio de dimensión n obteniéndose el n-cubo con n bits para cada vértice.

Una red n-cubo se corresponde con N nodos donde $n = \log_2 N$.

En un n-cubo cada nodo está conectado exactamente con n vecinos. Esos vecinos difieren exactamente en un bit.

Un cubo n-dimensional que conecta 2^n nodos con n >> 3 se denomina hipercubo.

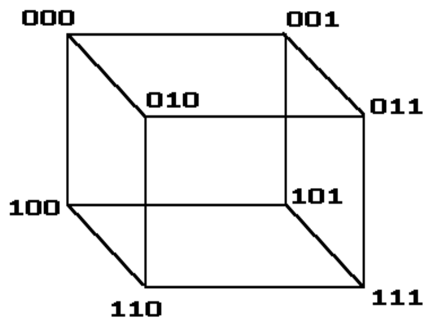


Fig. 7.10. - N-cubo de grado 3.

Las características principales de esta red son :

- La comunicación entre los procesadores es mediante caminos redundantes, luego, pueden ocurrir sobrecargas internas.

- Debido a las múltiples conexiones entre los nodos, las fallas en una rama no provocan la partición de la red.
- Para un cubo n-dimensional con 2^n nodos, la máxima distancia de comunicación entre dos nodos es **n** = $\log_2$ N.

Por ejemplo un cubo tridimensional requiere solamente 3 inputs por nodo para 2^3 procesadores. La cantidad de inputs crece con la cantidad de nodos, por ejemplo 256 nodos requieren 8 inputs para cada procesador.

En la Fig. 7.11 puede apreciarse la función de ruteo de datos entre los nodos de un n-cubo de grado 3.

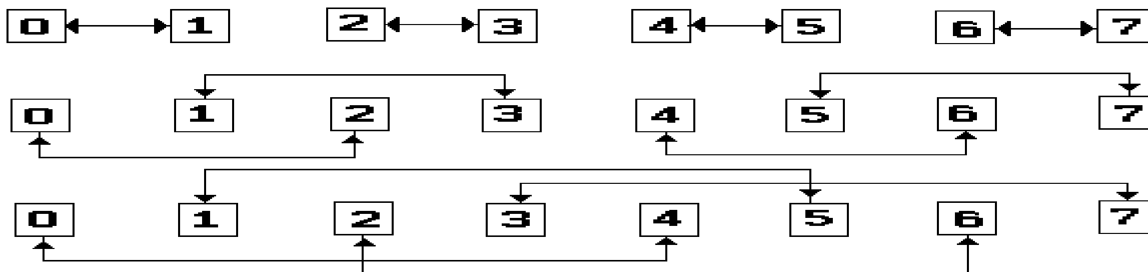


Fig. 7.11. - Función de ruteo de datos en un n-cubo de grado 3.

En la figura 7.12 se grafica un cubo cuatridimensional.

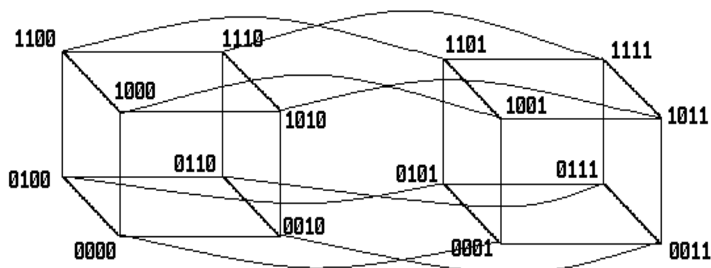


Fig. 7.12. - Hipercubo cuatridimensional.

7.4.8 - **Red Barrel Shifter**

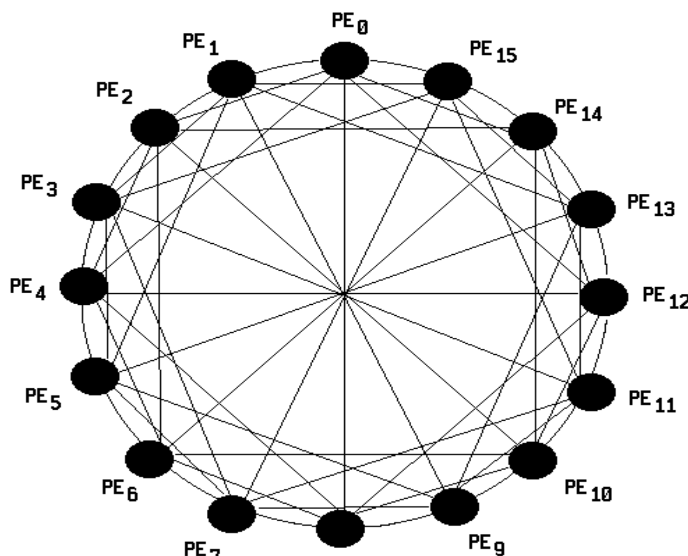
A título gráfico incluimos las redes Barrel Shifter conocidas también como redes PM2I (plus-minus-2^i).

En un Barrel Shifter de tamaño N con N = 2^n, se requieren B pasos para transmitir un mensaje de un nodo a otro, estando B acotado por :

B = $\log_2$ N / 2

La topología de red con Vecinos Cercanos es un subconjunto de las Barrel Shifter.

Por ejemplo, para N = 16 una topología con Vecinos Cercanos cuenta con 32 conexiones en tanto que un Barrel Shifter cuenta con 56. Véanse las figuras 7.13 y 7.14.



7.4.9. - **Arquitecturas con topología reconfigurable**

Si bien las arquitecturas de memoria distribuida conllevan a una topología física subyacente, las arquitecturas reconfigurables proveen conmutadores (switches) programables que permiten que el usuario seleccione la mejor topología lógica que se adecue a los patrones de comunicación que necesite (ver Fig. 7.15).

Algunas de estas máquinas se caracterizan

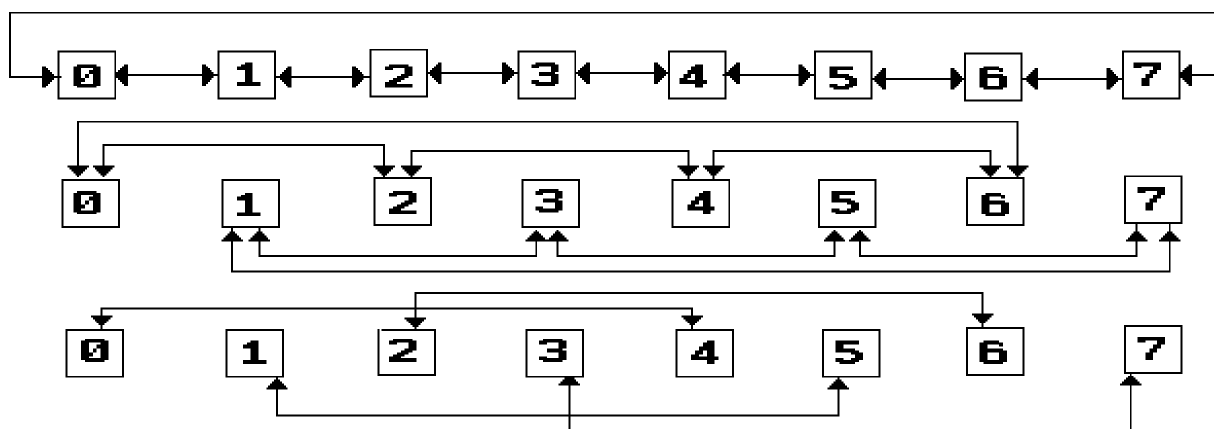


Fig. 7.14. - Función de ruteo de datos en un Barrel Shifter con 8 nodos.

por permitir definir distintas topologías lógicas (como en la Configurable Highly Parallel Computer o Chip de Lawrence Snyder) o por permitir el particionamiento de una topología en múltiples topologías del mismo tipo (como la Partitionable SIMD/MIMD System o Pasm de Howard J. Siegel).

El principal motivo para construir arquitecturas reconfigurables es el hecho de tratar de obtener que una arquitectura pueda actuar como muchas arquitecturas de propósito específico.

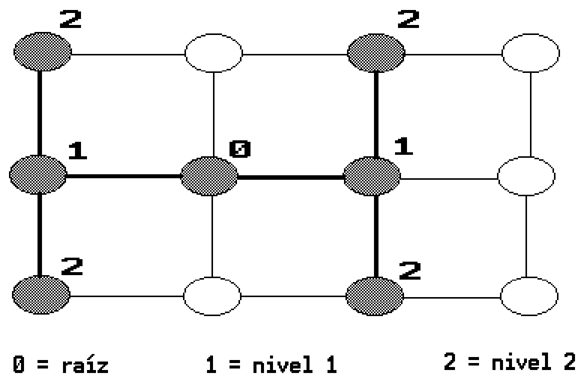


Fig. 7.15. - Una topología jerárquica mapeada sobre una red con vecinos cercanos de tipo reconfigurable.

7.5. - ARQUITECTURAS DE MEMORIA COMPARTIDA

Las arquitecturas de memoria compartida logran la coordinación entre los procesadores proveyendo una memoria global y compartida que cada procesador puede direccionar.

Estas arquitecturas tienen múltiples procesadores de propósito general que comparten la memoria, y no son del tipo CPU y procesadores de E/S.

Las computadoras de memoria compartida no tienen los problemas de las arquitecturas basadas en el pasaje de mensajes, como ser la latencia de envío del mensaje así como el encolamiento de datos desde y hacia los nodos.

Sin embargo, deben resolver problemas tales como la sincronización de acceso a los datos y la coherencia de la memoria cache.

La coordinación de los procesadores con variables compartidas requiere de mecanismos de sincronización atómicos para evitar que un proceso acceda a un dato antes de que el otro termine de actualizarlo.

Este mecanismo provee un "semáforo" que está sujeto al dato y que debe testearse antes de accederlo. El mecanismo "test-and-set" es un ejemplo de una operación atómica para controlar el valor del semáforo.

Generalmente cada procesador en una arquitectura de este tipo tiene una memoria local utilizada como cache. Sin embargo, pueden existir varias copias de la misma porción de memoria compartida en las cache de los procesadores en un momento dado.

Mantener una versión consistente de tales datos es el problema de la coherencia de la cache, el cual conlleva a que deben proveerse nuevas versiones a cada procesador cada vez que uno de los procesadores actualiza su copia.

A pesar de que los sistemas con un número pequeño de procesadores utilizan mecanismos hardware que "espían" las caches para determinar si han sido actualizadas, en los sistemas grandes se confía más en mecanismos de software para minimizar el impacto en la performance.

7.5.1 - Bus Multiacceso (Shared Bus)

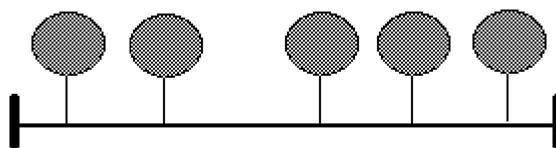


Fig. 7.16. - Bus multiacceso lineal.

En una red bus multiacceso existe una única conexión compartida : el bus. Todos los nodos están conectados a él, el cual puede estar organizado en forma lineal (Fig. 7.16) o en forma de anillo (Fig. 7.17). Se la conoce también como topología de Barra.

Los nodos pueden comunicarse a través del bus.

El costo básico de esta red es lineal respecto del número de nodos.

Efectivamente, un único bus de tiempo compartido se adecua bien para una cierta cantidad de procesadores (de 4 a 20), ya que uno solo de los procesadores accede al bus por vez.

Algunas arquitecturas basadas en bus como la arquitectura experimental Cm* utilizan dos tipos de buses: uno local que une un conjunto-cluster de procesadores y uno de sistema de más alto nivel que une los procesadores dedicados a servicios que se asocian a cada conjunto-cluster.

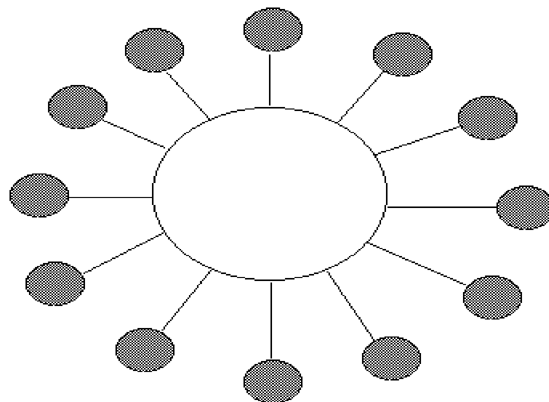


Fig. 7.17. - Bus multiacceso anillo.

El costo de comunicación es bastante bajo aunque la conexión puede convertirse en un cuello de botella.

Nótese que esta configuración es similar a la estrella salvo que aquí es un bus el que hace las veces de nodo central.

La falla de un nodo no afecta al resto. Sin embargo, si el que falla es el bus la red queda totalmente particionada.

7.5.2. - Crossbar Switch

El Crossbar Switch es un intento de superar el reducido troughput del sistema de bus multiacceso. El ejemplo más conocido es el crossbar switch del C.mmp.

En el C.mmp el crossbar switch conecta N procesadores con M módulos de memoria, pero también puede utilizarse para interconectar nodos en una red.

En cuanto a costo básico esta configuración es relativamente cara. Cuando la cantidad de procesadores es aproximadamente igual a la cantidad de memoria el costo es proporcional a N^2.

El crossbar switch provee una total conectividad con todos los módulos de memoria debido a que existe un bus separado para cada uno de ellos. Sin embargo, el máximo número de transferencias que pueden tener lugar simultáneamente está limitado a la cantidad de tales módulos.

Los conflictos se producen cuando se requieren dos o más solicitudes sobre el mismo módulo.

Los procesadores pueden competir para acceder a una ubicación en memoria, pero el crossbar impide la contención de las líneas de comunicación proveyendo un camino dedicado entre cada posible par de procesador/memoria.

Para proveer de más flexibilidad requerida en el acceso a los dispositivos de E/S, una extensión natural del crossbar switch es utilizar un switch similar en la parte de los dispositivos de E/S.

En la Fig. 7.18 puede verse este esquema conjuntamente con el esquema clásico para la interconexión de memorias-procesadores.

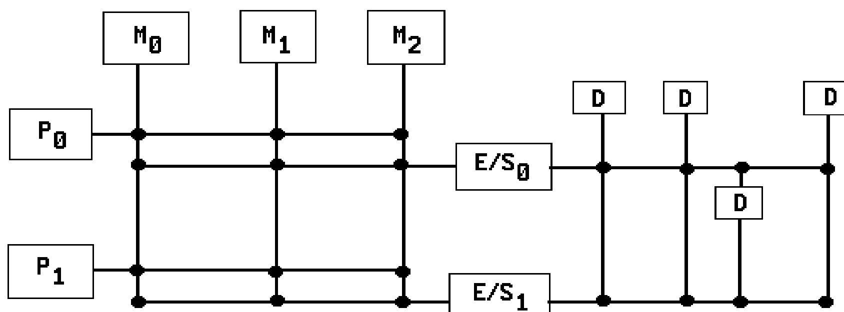


Fig. 7.18. - Crossbar Switch.

El crossbar switch es muy poderoso cuando existe un alto bandwidth.

La confiabilidad del switch es problemática; sin embargo puede mejorarse mediante segmentación y redundancia en el switch.

En general, es normalmente bastante fácil particionar el sistema para aislar las unidades lógicas que funcionan mal.

Cuesta trabajo justificar el uso del crossbar switch para grandes sistemas multiprocesadores, debido a la ausencia de un switch a un costo razonable y de buena performance.

El consumo de energía, la cantidad de patas de las conexiones (pinout) y ciertas consideraciones de tamaño han limitado las arquitecturas crossbar a un número pequeño de procesadores (de 4 a 16).

7.5.3. - **Redes de interconexión de múltiples etapas**

Las redes de interconexión de múltiples etapas (MIN- Multistage Interconnection Network) logran un compromiso entre las alternativas de precio/performance ofrecidas por los crossbar y los buses.

Una MIN de N*N conecta N procesadores a N memorias utilizando múltiples etapas o bancos de switches en el camino de la red de interconexión.

Cuando N es una potencia de 2, una forma es utilizar $\log_2 N$ etapas de N/2 switches, usando 2*2 switches

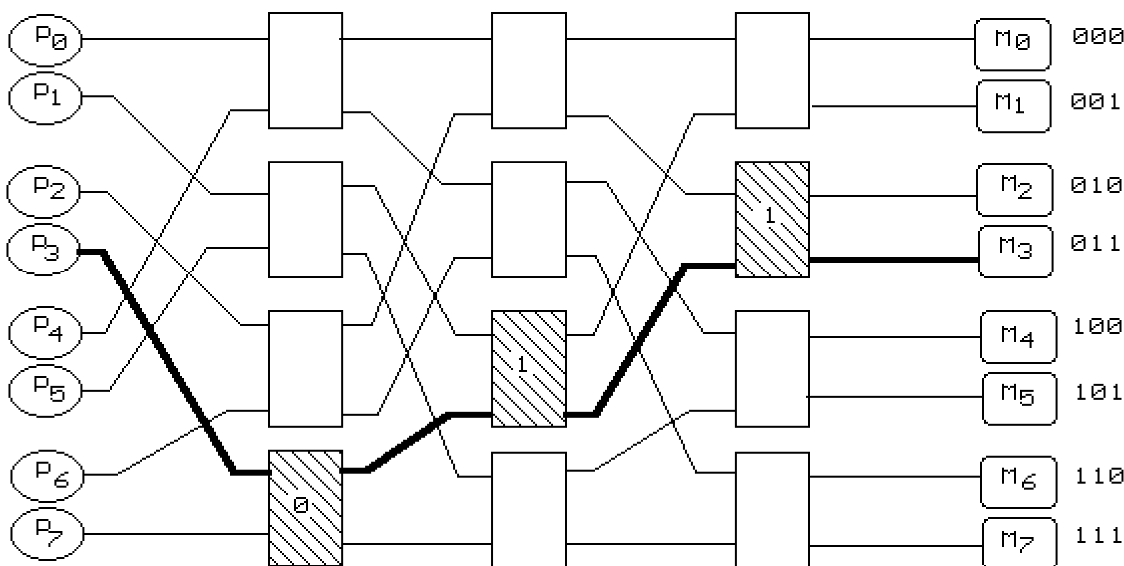


Fig. 7.19. - Ruteo de un requerimiento de P3 a M3 en una red Omega MIN de 8 * 8.

(ver Fig. 7.19). Un procesador que desea acceder a memoria hace un requerimiento especificando la dirección de destino (y el camino) mediante un vector de bits en donde cada uno es un bit de control para cada etapa.

El switch en la etapa i-ésima examina el i-ésimo bit para determinar si el pedido se rutea al output mayor o menor.

En la figura se puede ver una red omega que conecta 8 procesadores y memorias, cuando el bit de control está en cero indica que el requerimiento debe dirigirse al mayor output.

Una característica muy significativa de las redes MIN es que se pueden expandir, debido a que el diámetro de comunicación es proporcional a $\log_2 N$.

El Butterfly BBN (Blot, Neranek, y Newman) puede configurarse hasta abarcar 256 procesadores.

EJERCICIOS

1) Cuáles son las grandes razones para construir Sistemas Distribuidos ?
2) Qué es una topología de red ? Cuáles son los criterios utilizados en la materia para compararlas ?
3) Qué es el particionamiento de la red y cuándo sucede ?
4) Grafique e indique las características de las siguientes topologías de red :

- Totalmente conectada
- Parcialmente conectada
- Jerárquica o árbol
- Estrella
- Anillo
- Topologías reconfigurables
- Red con vecinos cercanos
- Hipercubo (grado 4)
- Barrel Shifter
- Bus multiacceso (shared bus)
- Crossbar switch
- MIN

5) Construya una topología de malla reconfigurable que mapee una estructura de anillo.
6) Construya la función de ruteo de datos en un Hipercubo de grado 4.
7) Cuanto pasos máximos se requieren para rutear un mensaje de un nodo a otro en un Barrel Shifter de 256 nodos ?
8) En qué consiste la similitud de una topología Estrella con una topología de Bus Multiacceso ?
9) Cuál es la característica de las direcciones de los nodos adyacentes en un hipercubo de grado **s** ? Cuántos nodos tiene este hipercubo ?

CAPITULO 8

TECNOLOGIAS RISC y CISC

8.1 - RISC

8.1.1 - Definición

La tendencia de hacer instrucciones de máquina cada vez más complejas, resulta en códigos de operación de tan alto nivel que tiende a romper la diferencia entre assembler y compilador.

Pero estudios al respecto revelaron elevados tiempos de diseño, aumento de errores, e implementaciones inconsistentes. Además se descubrió que las instrucciones muy complejas no eran utilizadas con mucha frecuencia y en muchos casos eran responsables del retardo de la performance de todo el sistema.

Esto último en particular debido a que la Unidad de Control debe ser más compleja y si es microprogramada provoca que todas las instrucciones demoren mayor tiempo en su ejecución.

Con el propósito de explotar nuevos caminos aparecen las tecnologías RISC (Reduced Instruction Set Computer) para oponerse a las CISC (Complex Instruction Set Computer) que son las que hemos visto hasta ahora.

En un intento de definir la tecnología RISC el grupo original de investigación en el proyecto RISC llevado a cabo en Berkeley (RISC I) produjo una filosofía de diseño que puede resumirse como sigue:

1) Analizar el objeto sobre el cual se desarrollan las aplicaciones a fin de determinar cuáles son las operaciones más frecuentes.
2) Optimizar los caminos que deben recorrer los datos para ejecutar las operaciones o instrucciones (del punto 1) tan rápido como sea posible.
3) Incluir otras instrucciones solo si forman parte de los caminos optimizados previamente y si son de relativa frecuencia, y si su inclusión no enlentece la ejecución de las instrucciones más frecuentes.
4) Aplicar una estrategia similar a los otros recursos del procesador. Incluir un recurso solo si éste está justificado por su frecuencia de uso, y su inclusión no enlentece a otros recursos más utilizados.
5) Tratar de trasladar lo más que se pueda la complejidad en tiempos de ejecución al momento de compilación, recargando el software de compilación y liberando al hardware de ejecución.

Algunas de las características más comunes que pueden verse en computadoras de tecnología RISC son:

- muchas de las instrucciones se ejecutan en un solo ciclo de máquina
- conjunto de instrucciones de cargar/almacenar (load/store). Es decir se accede a memoria exclusivamente mediante las instrucciones load y store, el resto de las instrucciones realizan sus operaciones entre registros
- decodificación de instrucciones hardwired (en oposición a la técnica de microprograma)
- existen relativamente pocas instrucciones y modos de direccionamiento
- todas las instrucciones deberían tener la misma longitud para facilitar la tarea de decodificación y homogeneizar los tiempos de carga de las mismas
- la complejidad se ha desplazado hacia los compiladores optimizados
- existe un alto grado de pipeline en los caminos de los datos para obtener mucha concurrencia
- gran cantidad de registros (windowed o no-windowed)
- muchos niveles de jerarquías de memoria
- conjunto de instrucciones diseñado para determinada clase de aplicaciones.
- soporte de lenguajes de alto nivel (Esto último debido a que como hay pocas instrucciones sobraría lugar en el chip para colocar elementos que ayuden a lenguajes de alto nivel, como manejo de listas, stacks, etc.)

Debe tenerse cuidado con una clasificación de computadoras de tecnología RISC, ya que, por ejemplo, muchas computadoras tienen decodificadores de instrucciones microcodificadas, un conjunto grande de instrucciones y un conjunto pequeño de registros pero sin embargo son de diseño definitivamente RISC.

La cuestión importante es que **esta filosofía RISC está respaldada en el diseño de un procesador para una aplicación específica.**

8.1.2 - Implementaciones de procesadores RISC

Los diseños RISC están disponibles como:

- microprocesadores de un único chip
- conjuntos de chips de muy alta escala de integración con funciones más poderosas
- computadoras de plaqueta única (single-board)
- superminicomputadoras

La mayor performance se está obteniendo de las implementaciones de arseniuro de galio (GaAs). Estos productos son los que tardarán más en llegar al mercado, los procesadores de GaAs tienen la ventaja de su inmensa velocidad.

Investigaremos primero el diseño de la CPU poniendo énfasis en el conjunto de instrucciones, decodificación de instrucciones, el camino de los datos, el diseño de registros, las unidades de ejecución, manejo de las bifurcaciones y el sistema de diseño de la memoria.

Luego investigaremos el RISC como un sistema, enfatizando las unidades de múltiple ejecución, el soporte de coprocesador, el multiprocesamiento, el soporte de sistema operativo y los lenguajes utilizados.
Finalmente compararemos performance.

8.1.3 - **CUESTIÓN CPU**

8.1.3.1 - **Conjunto de instrucciones**

El tamaño de los sets de instrucciones varía desde un mínimo de 16 instrucciones a aproximadamente 268.

Sin embargo el tamaño de la instrucción difiere en todas las máquinas RISC analizadas, todas utilizan formatos de instrucción que permiten una rápida decodificación utilizando un campo de código de operación consistente.

8.1.3.2 - **Decodificación de las instrucciones**

En este análisis existen dos diseños básicos de los decodificadores de instrucción y algunas combinaciones interesantes.

Recuérdese que los ejemplos de RISC antiguos utilizaban todos decodificadores de lógica hardwired por la posible facilidad y rapidez del decodificado de la instrucción.

Muchos de los procesadores utilizan algún tipo de decodificador de instrucción estrictamente por hardware o utilizan decodificación de tipo hardwired diseñada con una lógica minimizada. Otros hacen uso combinado de decodificación de tipo hardwired y microcódigo, algunos de estos casos tienen una unidad separada para obtención anticipada del código de operación (prefetch) y decodificación del mismo. La instrucción decodificada se carga luego en una memoria cache de instrucciones.

8.1.3.3 - **El camino de los datos (datapath)**

El diseño del datapath es del todo complejo en todos los procesadores analizados, tienen pipelines de una profundidad que varían desde 2 estadios (stages, etapas) hasta 7 estadios en los procesadores de arseniuro de galio.

En forma general, cuanto más corto es el tiempo del ciclo más profundo es el pipeline. Este fenómeno se debe a dos factores:

1º) todos los ejemplos de procesadores intentan comenzar a ejecutar una nueva instrucción cada golpe de reloj (clock cycle).

2º) los procesadores tienen accesos a memoria y retardos electrónicos que consumen una gran porción del período del ciclo de reloj.

Luego deben lograrse más accesos a memoria para obtener las instrucciones.

La Fig. 8.1 ilustra un pipe de 4 instrucciones de uno de los modelos de los procesadores RISC (el MIPS)

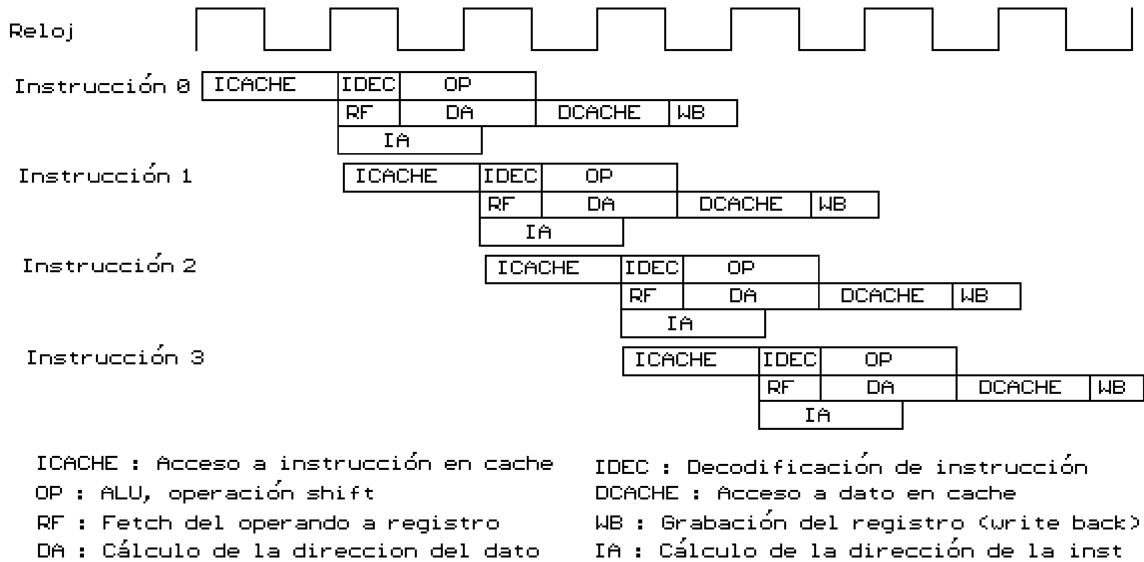


Fig. 8.1. - Un pipeline de 4 instrucciones de ejemplo.

cuyas instrucciones tienen todas una longitud de 32 bits.

Es bastante dificultoso mostrar una figura del número-de-etapas para este diseño debido a que existen estadios de medio-ciclo y ciclo-completo, y actividades concurrentes dentro de la ejecución de cada instrucción.

El ciclo de reloj es de 60 nanosegundos divididos en dos fases de 30 nanosegundos.

La complejidad está parcialmente justificada para ejecutar todas las instrucciones en un único ciclo.

8.1.3.4 - **Diseño de registros**

Se ha comprobado mediante mediciones que la mayoría de los procesadores CISC :

- gasta el 70 % de su tiempo accediendo a los operandos
- los operandos accedidos con más frecuencia son escalares
- cerca del 80 % de las referencias a escalares son locales al procedimiento
- el pasaje de parámetros representa alrededor del 10 % del total del tráfico de memoria

Y considerando que el porcentaje de llamadas a procedimientos ronda el 25 a 40 % del tiempo total de ejecución, esto sumado al ahorro que puede obtenerse en cuanto al manejo de las variables locales y al pasaje de parámetros indujo a que en todos los diseños RISC se intentara sacar provecho de la performance mediante:

1) la velocidad que se obtiene al almacenar variables dentro del chip
2) la habilidad del compilador para aprovechar en forma efectiva la propiedad de localidad de las variables de un programa

Generalmente existe, en consecuencia, una gran cantidad de registros que están organizados en forma de ventanas múltiples solapadas y de tamaño fijo.

Existen conjuntos de registros de ambas modalidades: windowed y no windowed.

La técnica de **windows de registros** permite que un nuevo conjunto de registros esté disponible para cada procedimiento, permitiendo una superposición de unos pocos registros para el pasaje de argumentos.

Cuando un procedimiento agotó la cantidad de windows, una de esas windows se libera salvando los datos que contiene en la memoria.

Si el dato fuera a ser requerido nuevamente se restaura en la window desde memoria.

Las condiciones en que una window es salvada o restaurada se denominan overflow y underflow respectivamente.

El hardware para windows está organizado como un buffer circular que cubre siempre la parte superior del stack de ejecución.

El microprocesador RISC I posee un total de 138 registros (numerados de 0 a 137), que utiliza de la manera que se aprecia en la Fig. 8.2.

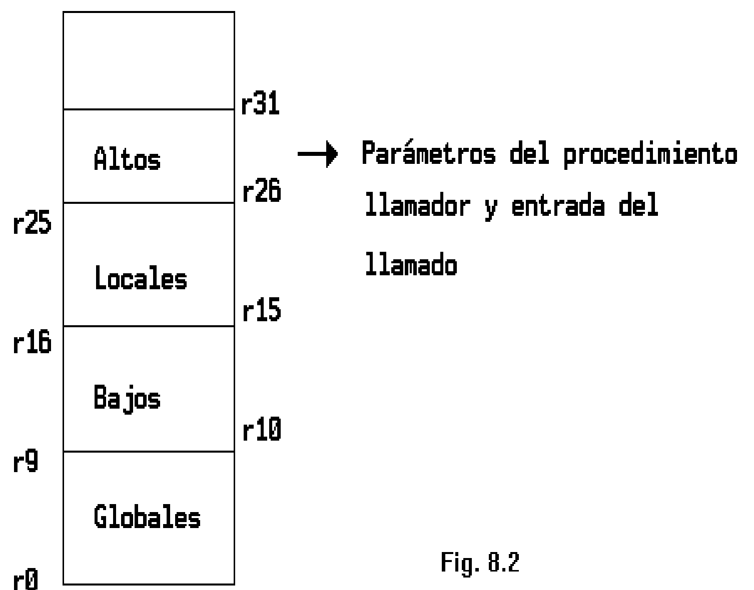


Fig. 8.2

Como sistema de ventanas los usa de la forma que puede visualizarse en la figura 8.3.

O sea el procedimiento llamador coloca sus parámetros de llamada en sus registros Altos, los cuales se transforman en Bajos para el procedimiento llamado, o sea que el procedimiento llamado antes de comenzar su ejecución ya tiene sus parámetros a disposición.

RISC I adoptó el sistema de 8 ventanas, pues con ese número de determinó que se producirían problemas de overflow sólo en el 1% de los casos.

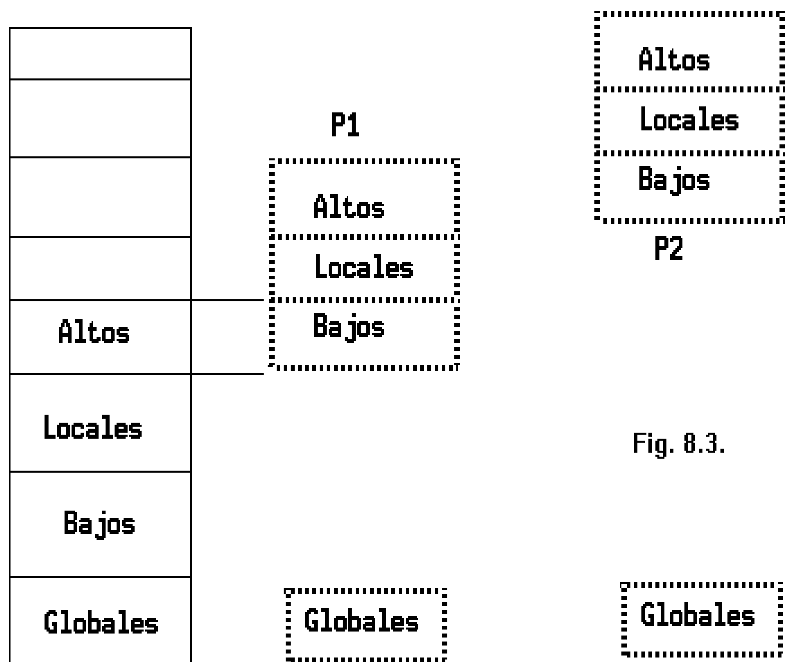


Fig. 8.3.

8.1.3.5 - **Unidades de ejecución**

Existen procesadores que tienen unidades separadas para la ejecución concurrente de operaciones ALU y de E/S, en varios de ellos la unidad de E/S tiene una ALU separada para cálculo de direcciones o tienen dos unidades

lógicas separadas, una para Prefetch y Decodificación y otra para Ejecución o coprocesadores para realizar tareas complejas tales como manejo de cache, administración de memoria y cálculos de punto flotante.

8.1.3.6 - **Manejo de las bifurcaciones**.

Varios estudios han demostrado que las ramas dentro de las bifurcaciones constituyen una fracción significativa del número de instrucciones ejecutadas.

En 1984 J. K. F. Lee y A. J. Smith realizaron un estudio sobre la cantidad de bifurcaciones en 26 programas totalizando una cantidad de 94 millones de instrucciones. El estudio consistió de una mezcla de programas de compilador, comerciales, científicos y de sistemas operativos corridos en una IBM S/370, programas educacionales corridos en una PDP11-70 y varios programas científicos corridos en una CDC 6400.

Esos programas mostraban que en promedio alrededor de un 10 a un 30 por ciento de las instrucciones en un programa típico son instrucciones de bifurcación, de las cuales la bifurcación es tomada en un 60 a un 70 por ciento de las veces. Esto último provoca que un pipeline de instrucción trabaje efectivamente al 66 % de su máxima capacidad.

El diseño de las instrucciones de salto es crucial en toda arquitectura aunque es de mucha más relevancia en arquitecturas pipelined. En realidad el elemento decisorio de este tipo de instrucciones se obtiene en tiempo de ejecución.

Existen procesadores que utilizan predicción sobre las bifurcaciones al momento de compilación, basado en asociar un bit que indica la probabilidad de que dicha bifurcación se produzca o no.

La técnica del "desdoblamiento" de las bifurcaciones usa un campo generado dinámicamente de "próxima instrucción" para cada instrucción decodificada.

Otros utilizan la técnica de los slots, mediante la cual se llenan unos o dos slots que continúan a la ejecución de la bifurcación con las instrucciones siguientes. Cuanto más niveles tiene el pipeline más dificultoso se torna llenar estos slots.

La tecnología RISC utiliza la técnica de prebúsqueda de instrucciones, pero como sabemos esto causa problemas con las instrucciones de salto, por lo tanto la técnica más utilizada es la de "salto demorado", de modo que el salto se realice luego de la siguiente instrucción (Ver Fig. 8.4).

Las máquinas con salto tradicionales ejecutan la secuencia de instrucciones (primer columna). Para obtener el mismo efecto en RISC, el compilador inserta una instrucción de no-operación (NOP) después de cada salto (segunda columna). Como el salto retardado asegura que la bifurcación se efectúa sólo después que las siguientes instrucciones han sido ejecutadas, los compiladores RISC incluyen frecuentemente un optimizador que intenta reemplazar este NOP con una instrucción que pertenezca al bloque precedente a la transferencia (tercera columna).

Las nuevas versiones de RISC ya utilizan la detección de salto incondicionales en la Unidad I.

	Salto Normal	Salto Retardado	Salto Optimizado
100	Load X, A	Load X, A	Load X, A
101	Add 1, A	Add 1, A	Jump 105
102	Jump 105	Jump 106	Add 1, A
103	Add A, B	Nop	Add A, B
104	Sub C, B	Add A, B	Sub C, B
105	Store A, Z	Sub C, B	Store A, Z
106		Store A, Z	

Fig. 8.4.

8.1.3.7 - **Sistema de Memoria**

Debido a la necesidad de retener las instrucciones y los datos que deben proveerse al procesador, los sistemas de memoria de los procesadores RISC son necesariamente muy complejos.

Se utilizan diversos niveles de jerarquía de memoria, muy a menudo con separación de datos e instrucciones.

Todos los procesadores analizados proveen el manejo de memoria virtual.

Por ahora tenemos diferentes estructuras de memoria, pero las más comunes incluyen un buffer de instrucciones dentro-del-chip de un tamaño suficiente para albergar unas pocas de las próximas instrucciones.

Este buffer se llena totalmente mediante una lógica de prefetch.

Algunos procesadores tienen una cache para datos y una cache para instrucciones, en tanto que otros solamente tienen una cache para todo o directamente carecen de ella.

Finalmente la memoria principal existe en construcciones fuera-del-chip y muy a menudo fuera del board del procesador.

De más está decir que el tiempo de acceso aumenta a medida que aumenta la distancia hasta la CPU o cuando aumenta el tamaño de la memoria.

8.1.4 - **División de funciones**

Casi todos los RISC cuentan con coprocesadores de punto flotante (uno o dos).

Algunas implementaciones de los multiprocesadores cuentan con :

- un array de varios procesadores RISC con memoria propia en un solo circuito. Una unidad de ejecución escalar obtiene las instrucciones y provee el control para este array.
- dos chips para manejo de cache y de memoria; uno maneja los accesos a datos a cache y a memoria, y el otro maneja los accesos a instrucciones en cache y memoria, esto permite por tanto superponer los accesos a datos e instrucciones.

8.1.4.1 - **Sistemas Operativos / Lenguajes Soportados**

El sistema operativo dominante es el UNIX con lenguajes: C, Fortran, Pascal, Ada y Cobol.

8.1.5 - **Notas de performance**.

Los procesadores en este panorama pueden dividirse en dos grupos de performance.

El mayor de los grupos contiene a aquellos procesadores con implementaciones de silicio.

En la Fig. 8.5 podemos ver que la mayoría de los procesadores tienen ciclos de reloj en el rango de 30 a 400 nanosegundos.

Procesador	Ciclo/Reloj	Promedio Instrucciones (MIPs)
Accel	100 ns	3.2 MIPS
ARM	8 MHz	3-4 MIPS
AMD2900	125 ns	4-5 MIPS
CAP	I : 10 MHz II : 25 MHz	12.5 MIPS maximo, unidad escalar
Clipper	33 MHz	5 MIPS
CRISP	16 MHz	> 10 MIPS
Dragon	10 MHz	5 MIPS por CPU
MIPS	16.6 MHz	8 MIPS
Pyramid	125 ns	2-4 MIPS
Ridge 32	125 ns	1-4 MIPS
ROMP	170 ns	2 MIPS
Spectrum	30 MHz	10.8 MIPS
Transputer	50 ns	10 MIPS
Whetstone	50 ns	5-13.3 MIPS
MIPS-X	20 MHz	> 10 MIPS
CD GaAs	5 ns	91 MIPS
McD GaAs	10 ns	100 MIPS
RCA GaAs	200 MHz	200 MIPS maximo

Fig. 8.5. - Performance de procesadores RISC.

El dato que figura en la columna de Promedio de Instrucciones es, en la medida de lo posible, el promedio de tiempos de procesamiento de mediciones o benchmarks simulados, y no representa los valores pico.

Los 10 MIPs del Transputer pueden parecer engañosos, ya que se basan en el supuesto de que todas las instrucciones y operandos residen en el chip de RAM sin que existan demoras de acceso a memorias externas.

El segundo grupo de performance incluye los procesadores de arseniuro de galio.

Los tres procesadores fueron diseñados para un ciclo de reloj de 200 Megahertz. Los promedios de ciclo y de instrucciones son de una magnitud mayor que la de aquellas implementaciones basadas en el silicio.

Pocos de los procesadores vistos poseen cada una de las características atribuidas a los diseños RISC.

Muchos comparten alguna de las características de las CISC, agregando capacidad de procesamiento adicional para una dada aplicación.

De hecho, es interesante hacer notar que cada procesador de los analizados contiene características arquitecturales atribuidas típicamente a los CISC, y que cada característica de las CISC está representada en al me-

nos alguno de los diseños RISC, indicando que en el futuro los buenos diseños y conceptos arquitectónicos útiles sobrevivirán.

Es desde ya obvio que un compilador optimizado es una parte integral de cualquier diseño RISC.

El desarrollo de compiladores optimizados y reorganizadores va a la zaga muy a menudo de los nuevos desarrollos de hardware de cómputo.

8.2. - **CONTROVERSIA**

Existe una gran controversia cuando se quiere comparar computadoras con conjunto de instrucción reducido y computadoras con conjunto de instrucción complejo (RISC vs. CISC). Esta misma controversia puede dividirse en dos grandes categorías:

1º) Qué diferencia una RISC de una CISC ?

2º) Cómo puede uno hacer mediciones razonables y útiles de performance para compararlas ?

Muchas de las características de las RISC han sido muy utilizadas en computadoras CISC. Características tales como el pipeline de datos, memoria cache, y windowing de registros son vistos muy a menudo como atributos de un diseño RISC.

Originalmente los diseños RISC se realizaron apuntando a aplicaciones específicas y debido a eso fueron optimizados para la ejecución de una clase bien definida de programas.

De forma característica, las CISC fueron diseñadas para un amplio rango de aplicaciones y consiguientemente incluyen el soporte para muy diversos entornos de programación.

Muchas de las más populares técnicas de medición de performance son de un valor cuestionable cuando se trata de medir performance entre RISC y CISC. Típicamente, los efectos del overhead del sistema operativo, la optimización del compilador y los conjuntos de registros múltiples no son considerados en forma apropiada.

Los benchmarks relativos a la cantidad de transacciones de la aplicación por segundo tienen más significado que la simple medición de las instrucciones ejecutadas por segundo.

Actualmente aún existen fuertes disputas entre los defensores de RISC y CISC. De acuerdo al avance del mercado se detectan RISC sólo de nombre (ya que poseen más de 180 tipos diferentes de instrucciones) pero que mantienen la llamada a procedimientos desde hardware y lo realmente novedoso que es el sistema de ventanas.

Existen ya versiones que realizan operaciones de punto flotante en el llamado ciclo de máquina.

Se define el tiempo de ciclo de máquina como el tiempo que lleva leer y sumar el contenido de 2 registros y guardar el resultado en un tercero.

8.3 - **Una definición de las arquitecturas CISC**

Los criterios que siguen a continuación pueden utilizarse tanto para definir computadoras RISC como computadoras CISC.

Aquí los utilizaremos para las computadoras CISC, con las precisiones correspondientes en cada caso:

1) Cantidad de instrucciones en lenguaje máquina (en el caso de las CISC, tan grande como sea posible)
2) Cantidad de modos de direccionamiento (aquí también tan grande como sea posible)
3) Cantidad de formatos de instrucción (nuevamente, tan grande como sea posible)
4) Muchas instrucciones requieren más de un ciclo para su ejecución
5) Varios tipos de instrucciones tienen acceso a memoria (en el caso de los sistemas RISC las instrucciones LOAD/STORE son las únicas que tienen acceso a memoria)
6) Existencia de registros de propósito específico
7) Control microprogramado
8) Instrucciones de máquina de un relativo alto-nivel (cercano al alto nivel de las sentencias de los lenguajes de alto nivel)

Los criterios (1) y (2) pueden especificarse en valores numéricos, en tanto que los otros se especifican por Si o No.

Asumamos los siguientes requerimientos a efectos de cumplir los criterios (1) y (2):

1) la cantidad de instrucciones debe ser mayor a 100
2) los modos de direccionamiento son más de 4

La Fig. 8.6 presenta los ocho criterios para las siguientes arquitecturas CISC:

- El Motorola MC68020
- Intel 80386
- Clipper de Fairchild
- Zilog Z80000
- AT&T WE32100
- Focus de Hewlett-Packard
- Serie NS32000
- DEC VLSI de VAX

La Fig. 8.7 indica, para cada máquina en particular si los 8 criterios han sido satisfechos (S) o si han sido violados (N).

Según surge de las Fig. 8.6 y 8.7 muchos de los procesadores satisfacen muchos de los requerimientos y pueden ser caracterizados entonces como máquinas CISC.

SISTEMA	Criterios							
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Motorola 68020	109	16	Si	Si	Si	Si(16)	Si	Si
Intel 80386	111	8	Si	Si	Si	Si(6)	Si	Si
Clipper	101	9	Si	No	No	No(16)	No	No
Zilog Z80,000	110	9	Si	Si	Si	No(16)	Si	Si
AT&T WE32100	169	16	Si	Si	Si	No(16)	Si	Si
Focus de HP	230	10	Si	Si	Si	Si(28)	Si	Si
Series NS32000	86	14	Si	Si	Si	No(8)	Si	Si
DEC VLSI VAX	304	21	Si	Si	Si	Si(16)	Si	Si

* Bajo ciertas condiciones los valores numéricos de esta tabla pueden variar

Fig. 8.6. - Ocho arquitecturas evaluadas en términos de los criterios que caracterizan a las computadoras CICS.

SISTEMA	Criterios							
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Motorola 68020	Si	Si	Si	Si	Si	Si	Si	Si
Intel 80386	Si	Si	Si	Si	Si	Si	Si	Si
Clipper	Si	Si	Si	No	No	No	No	No
Zilog Z80,000	Si	Si	Si	Si	Si	No	Si	Si
AT&T WE32100	Si	Si	Si	Si	Si	No	Si	Si
Focus de HP	Si	Si	Si	Si	Si	Si	Si	Si
Series NS32000	No	Si	Si	Si	Si	No	Si	Si
DEC VLSI VAX	Si	Si	Si	Si	Si	Si	Si	Si

Fig. 8.7. - Ocho arquitecturas evaluadas en términos de si satisfacen (Si) o violan (No) los criterios que caracterizan a las computadoras CISC.

La única excepción la constituye el Clipper de Fairchild, el cual combina características de máquinas CISC (gran cantidad de modos de direccionamiento y formatos de instrucciones) y características de máquinas RISC (incluyendo ejecución de instrucción en un solo ciclo, arquitectura LOAD/STORE y control de tipo hardwired).

Algunas de las características arquitecturales son erróneamente consideradas como típicamente de computadoras CISC; como por ejemplo, la gran cantidad de registros, el soporte para la administración de memoria y de memoria virtual, la existencia de una memoria cache, y una gran cantidad de transistores en el chip.

Estas características arquitecturales están reflejadas en la Fig. 8.8.

La columna (a) indica dónde está ubicada la unidad de administración de memoria, la columna (b) define dónde está ubicada la memoria cache, y la columna (c) especifica en forma conjunta la cantidad de transistores y la cantidad de chips utilizados en la construcción del procesador.

Los diseñadores de máquinas CISC y RISC, tienden a incorporar una gran cantidad de registros en sus máquinas.

Sin embargo, en el caso de las RISC tales registros tienden a ser de propósito general, en tanto que en el caso de las CISC una cierta cantidad de los registros son de propósito específico. Este hecho se ve reflejado en las Fig. 8.6 y 8.7.

Es cierto que las CISC están caracterizadas típicamente por una gran cantidad de transistores dentro del chip, a diferencia de las RISC. Sin embargo, una RISC con una gran cache dentro del chip puede tener incorporados una cantidad aún mayor de transistores.

La existencia de una memoria cache y del soporte para la administración de memoria y de memoria virtual son ambas, de una importancia comparable para las CISC y las RISC. Sin embargo, es cierto, que muchas (aun-

que no todas) las máquinas RISC proveen la administración de memoria fuera del chip en tanto que la mayoría de las CISC intentan incorporar al menos algunos de los elementos de la administración de memoria dentro del chip.

SISTEMA	Características de Arquitectura		
	(a)	(b)	(c)
Motorola 68020	Fuera del chip	Dentro del chip	190 K (1)
Intel 80386	Dentro del chip	Fuera del chip	275 K (1)
Clipper	Fuera del chip	Dentro del chip	(3) *
Zilog Z80,000	Dentro del chip	Dentro del chip	(1) *
AT&T WE32100	Fuera del chip	Dentro del chip	146 K (3)
Focus de HP	Fuera del chip	Dentro del chip	450 K (1)
Series NS32000	Dentro del chip	Dentro del chip	(1) *
DEC VLSI VAX	Dentro del chip	Fuera del chip	1.2 M (9)

* Datos no disponibles en la literatura

Fig. 8.8. - Ocho arquitecturas se evaluan aquí en términos de sus características arquitecturales.

EJERCICIOS

1) Cómo puede definirse la tecnología RISC y cuáles son sus características más comunes ?
2) Comente la controversia existente sobre las diferencias entre procesadores RISC y CISC.
3) Porqué la mayoría de los procesadores RISC tiene una lógica de decodificación de tipo hardwired ?
4) Verdadero o falso : Muchos procesadores RISC tienen unidades separadas para la ejecución concurrente de operaciones aritméticas y de E/S.
5) Qué es el windowing de registros en los procesadores RISC ?
6) Cómo se manejan las bifurcaciones en los procesadores RISC ? Comente cómo cree que se implementa.
7) Porqué existen coprocesadores en la mayoría de los procesadores RISC ?
8) Comente si la siguiente frase le parece o no razonable : "Dado un problema específico, es menos eficiente resolverlo con un procesador RISC que en una CISC de propósito general".
9) El uso de registros en los procesadores RISC es de uso general o específico ?
10) Los procesadores RISC tienen unidades de control microprogramadas o hardwired ? Justifique.
11) Porqué es usual que los procesadores RISC cuenten con pipelines de instrucción ?
12) Porqué resulta importante el desarrollo de los compiladores para máquinas RISC? Justifique.
13) Comente por lo menos tres criterios que usted utilizaría para diferenciar una computadora CISC de una RISC.
14) Cuál es la diferencia entre cómo se utilizan los registros internos de la CPU en computadoras RISC y CISC ? Utilizaría este concepto para diferenciar estas arquitecturas ?

CAPITULO 9

ARQUITECTURAS NUEVAS

9.0 - **Breve introducción**

Como habíamos anticipado en el final del capítulo 6 existen una serie de arquitecturas difíciles de acomodar en una clasificación genérica de arquitecturas paralelas, ellas eran, a saber :

- Arquitecturas Dataflow
- Arquitecturas híbridas MIMD/SIMD
- Arquitecturas de Reducción
- Arquitecturas de Wavefront Array

En este capítulo veremos en más detalle tales arquitecturas poniendo especial énfasis en las máquinas Dataflow debido a la filosofía en que se basan.

9.1 - **Introducción a DATAFLOW**

Uno de los objetivos primordiales en el desarrollo de los sistemas es alcanzar la más alta velocidad y performance posibles. Este objetivo ha sido y sigue siendo alcanzado por dos medios :

- explotando las posibilidades tecnológicas de los componentes del computador, y
- adecuando estructuras y organizaciones del computador.

El intento de producir computadoras de arquitectura paralela con alta velocidad y performance obedece a las necesidades de resolver grandes problemas (la mayoría de características paralelas) tales como reconocimiento de patrones (pattern recognition), procesamiento de señales e imágenes (signal & image processing), problemas de inteligencia artificial, física nuclear, predicción meteorológica, control de tráfico aéreo, procesos de control de producción, etc. ; muchos de los cuales deben resolverse en tiempo real.

Computadoras tales como la ILLIAC IV y computadoras asociativas como la STARAN tienen una muy alta performance del orden de 10^2 MIPS para datos de una estructura adecuada o grandes sistemas de vectores en los cuales la misma operación se produce simultáneamente.

Su eficiencia para resolver problemas en donde la estructura de datos no es tan regular disminuye rápidamente, aún cuando son naturalmente paralelas. Estas computadoras son absolutamente ineficientes para procesar problemas seriales.

Otra desventaja substancial es la gran dependencia de la performance de estas computadoras del método de programación del problema. El programador debe conocer la estructura interna del computador para ser capaz de poder utilizar su paralelismo, y debe él mismo identificar el paralelismo en el problema o utilizar un programa especial para hacer esto último.

Muchos autores han llegado a la conclusión de que el problema central de la utilización del paralelismo reside en el modelo básico para expresar y describir el mismo.

El modelo de cómputo de Von Neumann aparece inapropiado a este respecto debido a ser un modelo serial.

Sorprendentemente todos los lenguajes de programación convencionales y las arquitecturas de las computadoras existentes, tanto SISD, SIMD, o MIMD están basados en este modelo de cómputo.

La naturaleza serial del modelo de Von Neumann y los problemas resultantes de ello son el principal obstáculo en la utilización del paralelismo en las computadoras paralelas que procesan programas escritos en lenguajes convencionales de alto nivel.

Luego, se hace necesario crear primero un modelo de sistema computador el cual permita expresar algoritmos de un paralelismo natural.

Uno de tales modelos es el modelo Dataflow conocido como sistema Data-Driven. Estrechamente relacionado con esto está el trabajo sobre utilización del principio de asignación única para expresar el paralelismo.

9.1.1 - **El modelo de cómputo Dataflow**

Cuáles son las diferencias entre el modelo de cómputo Dataflow (DF) y el modelo convencional de cómputo Von Neumann (Control flow, CF) ?.

En principio, la diferencia radica en qué es lo decisivo en el proceso de cómputo en cada modelo:

- en el CF, es la secuencia de las instrucciones.
- en el DF, es la disponibilidad de los datos.

En una computadora convencional CF, el programa se almacena en la memoria como una secuencia de instrucciones. El programa se ejecuta extrayendo las sucesivas instrucciones desde la memoria y ejecutándolas en el procesador.

Luego, el curso que sigue el cómputo está dado por la secuencia de las instrucciones del programa (es decir, el flujo de control del programa).

No es posible ejecutar cualquier instrucción hasta que todas las instrucciones previas hayan sido ejecutadas. Si los operandos necesarios están disponibles, pueden existir en el programa instrucciones que podrían ejecutarse ya, pero ellas deben esperar a que les toque su turno en la secuencia.

Este es el obstáculo principal en la utilización de algoritmos de un paralelismo natural.

En las computadoras DF, el curso del cómputo está controlado por el flujo de los datos en el programa. Una instrucción puede ejecutarse solamente cuando están disponibles todos sus operandos.

Estos operandos pueden ser obtenidos como datos de entrada o por ser resultado de instrucciones previas en el programa.

La precedencia de una instrucción respecto de otra está dada aquí exclusivamente por la estructura natural del algoritmo que se está realizando y no depende de la ubicación de las instrucciones en la memoria.

Utilizando este modelo de cómputo es posible llegar a ejecutar tantas instrucciones en paralelo como pueda el computador en cuestión.

Luego de la ejecución de la instrucción el resultado se distribuye a todas las instrucciones subsiguientes que hagan uso de ese resultado parcial como operando.

9.1.2 - **Representación de programas Dataflow**

El diseño del modelo de cómputo DF se explica mejor con un ejemplo concreto.

Una de las mejores representaciones de un programa dataflow es mediante un grafo dirigido el cual puede ser visto como una simplificación del lenguaje DF.

Los nodos en el grafo representan operadores (instrucciones de programa) y los arcos representan caminos de datos unidireccionales a través de los cuales los resultados se transmiten en el programa.

En general, es posible representar la ejecución de una instrucción como una función sobre n inputs con m outputs, como se ve en la Fig. 9.1.

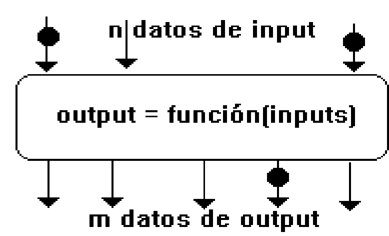


Fig. 9.1. - Una representación simbólica del cómputo DF.

Un punto negro sobre el camino del dato representa la presencia del operando apropiado o del resultado parcial. Una instrucción sólo puede ejecutarse cuando todos sus inputs contienen operandos.

El operador "consume" sus inputs y libera un conjunto de resultados sobre los caminos de output.

9.1.3 - **Tipos de Operadores**

Muchos de los lenguajes dataflow están limitados a dos tipos fundamentales de instrucciones (combinativa y separativa) con una cantidad de inputs y outputs iguales a 1 o 2 (Fig. 9.2).

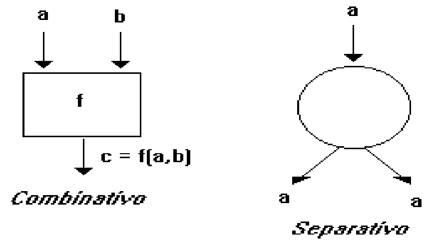


Fig. 9.2. - Los tipos básicos de operadores Dataflow.

Las instrucciones combinativas tienen dos inputs y un output. El valor del output es una función de los valores de los inputs. Este operador puede ejecutar funciones sencillas como la suma, OR,

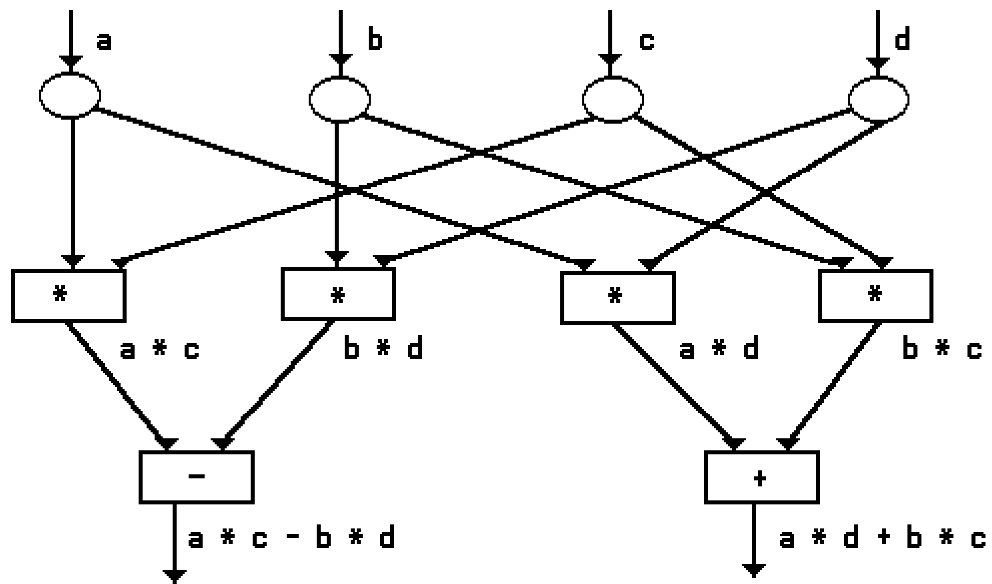


Fig. 9.3. - Un programa sencillo dataflow para multiplicar dos números complejos (a+bi) * (c+di).

AND, etc. o funciones complejas tales como la multiplicación, división, o incluso procedimientos completos.

Las instrucciones separativas producen dos copias del dato de input que pueden ser de tipo diferentes.

Los caminos de los datos representan el flujo de los datos desde una instrucción hacia la próxima. Este es el sistema de comunicación de la ingeniería de implementación. Este sistema de comunicación puede ser un simple bus del sistema, pero puede también almacenar datos en memoria utilizando un sistema FIFO.

Como ejemplo de un cálculo DF se muestra en la Fig. 9.3 la ejecución de un programa sencillo que simula la multiplicación de dos número complejos.

La Fig. 9.4 muestra la ejecución de este programa sobre una computadora DF en cuatro fases diferentes, a saber :

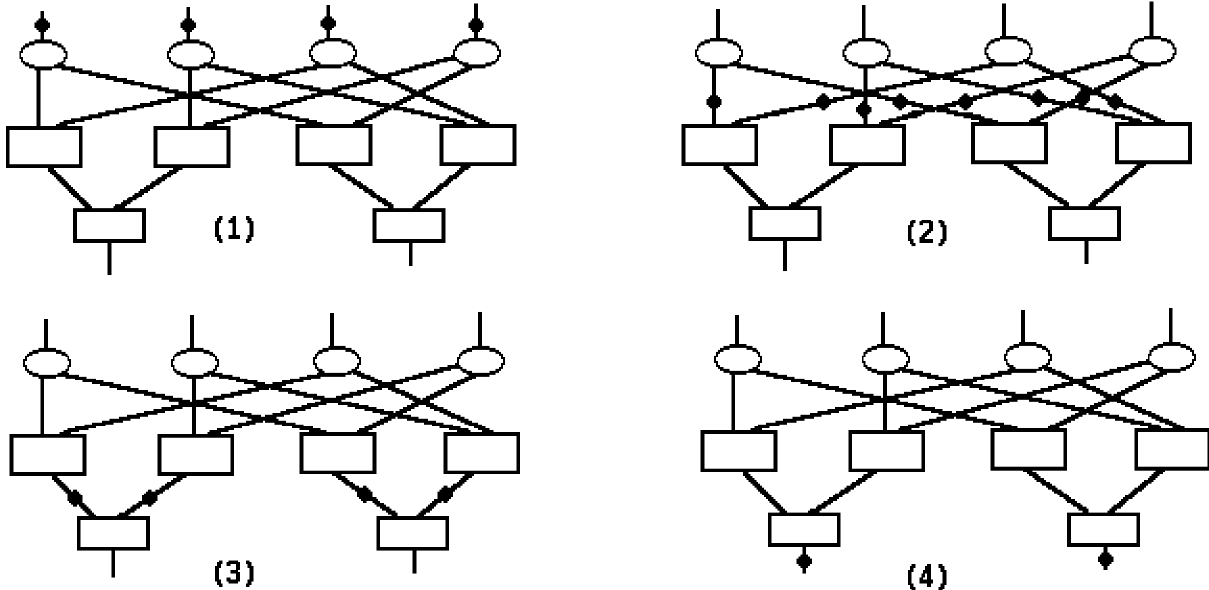


Fig. 9.4. - La ejecución del programa de la multiplicación de dos números complejos.

(1) el estado antes del comienzo del cálculo : todos los operandos están presentes y los inputs de las cuatro funciones están cargados.
(2) estado luego del primer ciclo : se producen copias de los operandos aislados y se distribuyeron a los operadores que los utilizarán.
(3) estado luego del segundo ciclo : los productos parciales ad, bc, ac, y bd se han obtenido y se distribuyen a los siguientes operadores.
(4) estado luego del tercer ciclo : se ejecutaron la suma de ad con bc y la resta de bd a ac ; estos dos outputs forman el resultado complejo.

El estado del cálculo está representado por puntos en los caminos de los datos representado tokens, los que corresponden tanto a variables de input como a resultados parciales. Mediante estos tokens es posible representar la dinámica del cómputo.

9.1.4 - **TOKENS**

Pero la palabra Token es un término aún más amplio :

Token *: es el valor de una variable sobre un camino de dato la cual representa el resultado actual de una operación que ha sido realizada por el operador precedente.*

Un token puede contener dos tipos de datos :

- el dato : uno o varios operandos que pueden ser de diferentes tipos : entero, real, booleano, string, incluso estructuras más complejas como procedimientos completos que pueden dibujarse con grafos de flujos separados ;
- dato de control : el código de operación, direcciones de los resultados, señales, código de programa, cantidad de iteraciones, etc.

El contenido real y el formato de cada token depende del lenguaje del programa DF concreto y de la arquitectura del computador.

Un token es una unidad independiente completa. En principio, normalmente contiene el dato y no direcciones de memoria. No contiene referencias a otros tokens y unidades comunes del computador.

Luego, un token nos dice :

QUE se debe hacer - un token contiene el código de operación.

DONDE debe entregar el resultado - un token contiene la dirección en donde debe entregarse el resultado. Este mecanismo asegura la continuidad del programa.

CUANDO un cálculo debe ejecutarse - un método para sincronización. El cálculo se realiza solamente cuando un token en el input del operador está completo.
CON QUE se ejecuta el cálculo - el token contiene operandos sobre los cuales la instrucción dada se ejecutará.

9.1.5 - **Estados de los operadores**

El operador puede estar en uno de los cuatro siguientes estados (Fig. 9.5), a saber :
- ocioso, el operador tiene por lo menos un token (en el caso de ser un operador combinativo) o ningún token (de ser separativo) en el input y ningún token en el output ;
- habilitado, todos los tokens de inputs requeridos están presentes y está listo para ejecutar la función;
- activo, está ejecutando la función ;
- generando, genera el resultado, un nuevo token, y lo entrega al próximo operador.

En la literatura, se han mencionado hasta ahora dos posibles implementaciones :
- el operador no debe pasar desde el estado de habilitado al estado activo mientras exista un token previo en su output;
- el operador puede pasar del estado habilitado al estado activo prescindiendo de que tenga uno o varios tokens previos en su output. En este caso, pueden estar presentes colas de tokens en los caminos de los datos.

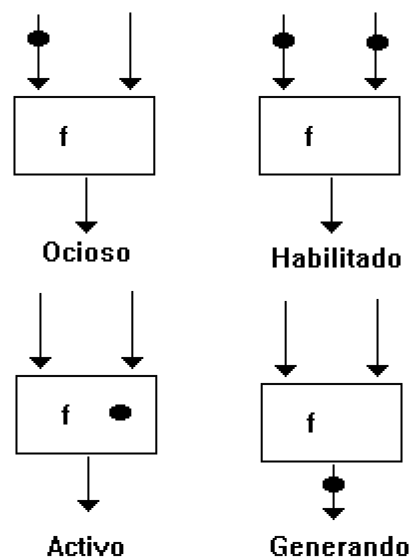


Fig. 9.5. - Estados de un operador Dataflow.

9.1.6 - **Intérpretes Feedback y No-Feedback**

El asunto de la generación de colas en los caminos de los datos es la cuestión clave para los lenguajes de los programas DF, la arquitectura de los computadores DF y la organización de las acciones en las computadoras DF.

En el primer caso, cuando el operador debe esperar hasta que su output haya sido limpiado, es necesario que el procesador pueda representar la información mediante la cual un operador pueda informar a los operadores precedentes el hecho de que ya ejecutó el cálculo actual, es decir, que ya ha consumido el token.

La acción de una computadora de este tipo está basada en un intérprete feedback FI (o de retroceso).

Una acción en la cual el operador puede generar tokens independientemente de si los tokens precedentes han sido ya consumidos o no, está basada en un intérprete no-feedback, NFI.

Esta acción supone la posibilidad de almacenar series de resultados parciales, correspondientes a los tokens, durante la transición de un operador a otro.

Como regla, la memoria trabaja en un principio FIFO.

Existen también extensiones del concepto previo en las cuales una de las series generada de tokens no se almacena en el sistema FIFO, pero puede ser extraída de la cola en un orden arbitrario, por ejemplo, tokens aislados pueden "saltar de la cola", lo que permite una aceleración de la velocidad del cálculo debida a una mejor utilización de los varios niveles de paralelismo.

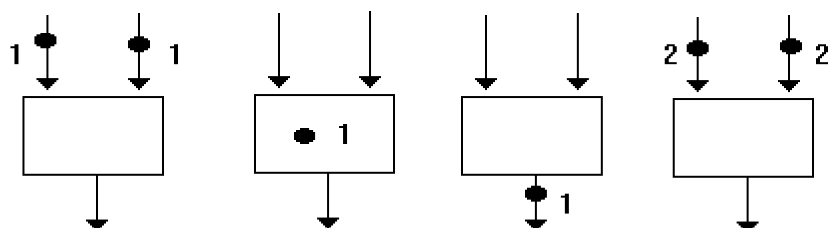


Fig. 9.6. - La ejecución de un cálculo bajo un intérprete feedback.

El camino de los datos puede estar en alguno de los siguientes estados :
- cargado en FI : uno y solo un token están presentes en él;
- cargado en NFI : uno o más tokens

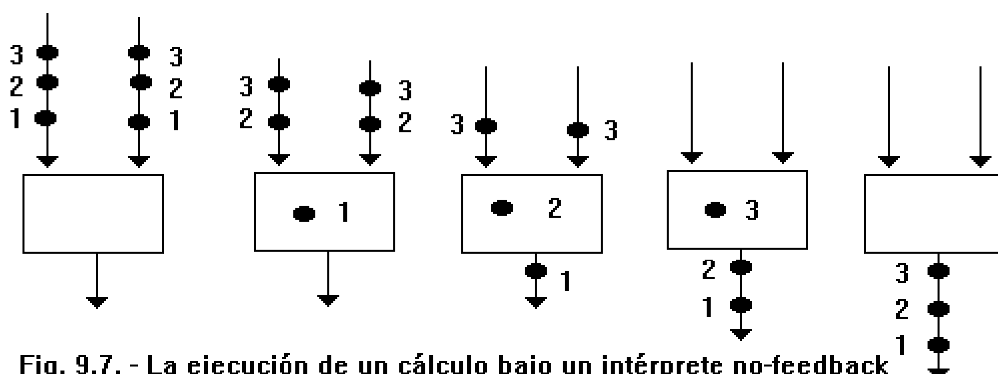


Fig. 9.7. - La ejecución de un cálculo bajo un intérprete no-feedback con encolamiento FIFO de tokens.

están presentes en él.

- descargado - ningún token está presente en él.

Para una mejor comprensión del comportamiento bajo intérpretes FI y NFI mostramos los siguientes tres ejemplos :

Fig. 9.6 muestra el curso de un cálculo bajo FI.

Fig. 9.7 muestra el curso de un cálculo bajo NFI conjuntamente con encolamiento de tokens FIFO en los caminos de datos.

Fig. 9.8 muestra un cálculo bajo NFI conjuntamente con encolamiento arbitrario de los tokens sobre el camino de datos.

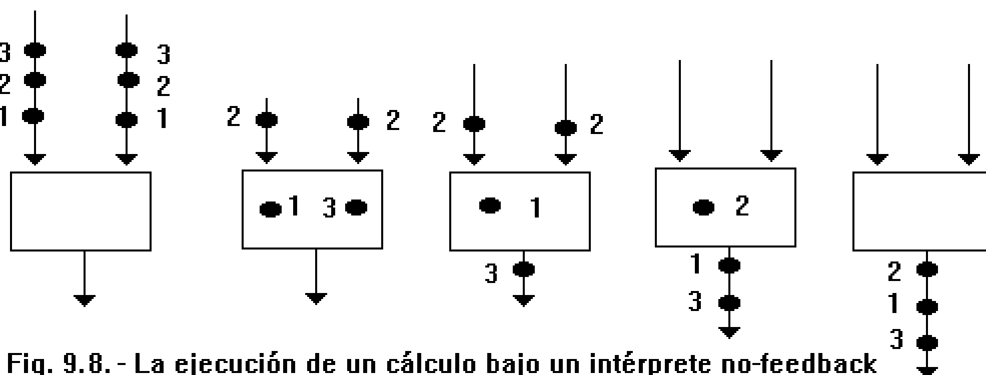


Fig. 9.8. - La ejecución de un cálculo bajo un intérprete no-feedback con encolamiento arbitrario de tokens.

Nótese en la Fig. 9.8 que en el segundo ciclo ocurre un procesamiento simultáneo de dos tokens.

Esto no significa que en el computador ellos hayan sido procesados por el mismo procesador simultáneamente, sino más bien que fueron procesados por diferentes procesadores al mismo tiempo.

El token 2 pudo ser "superado" por el token 3 por varias razones. Por el momento, el procesador puede estar especializado en el procesamiento de estructuras de datos particulares y el dato en el token 3 puede procesarse más rápidamente, o el volumen de los datos en los tokens pueden ser distintos.

Recalcamos que la estructura de un token puede ser bastante compleja.

9.1.7 - **ARQUITECTURAS ESTATICAS Y DINAMICAS**

Las arquitecturas Dataflow pueden dividirse en Estáticas y Dinámicas

9.1.7.1 - **Arquitecturas Estáticas**

En este tipo de arquitecturas los data token se desplazan por los arcos del grafo del programa y, la operación se ejecuta cuando todos los datos (token) están presentes en los arcos de entrada de los operadores.

Sólo se permite que un solo token esté presente en cualquier arco en un instante dado, sino el conjunto de tokens **no** podría distinguirse.

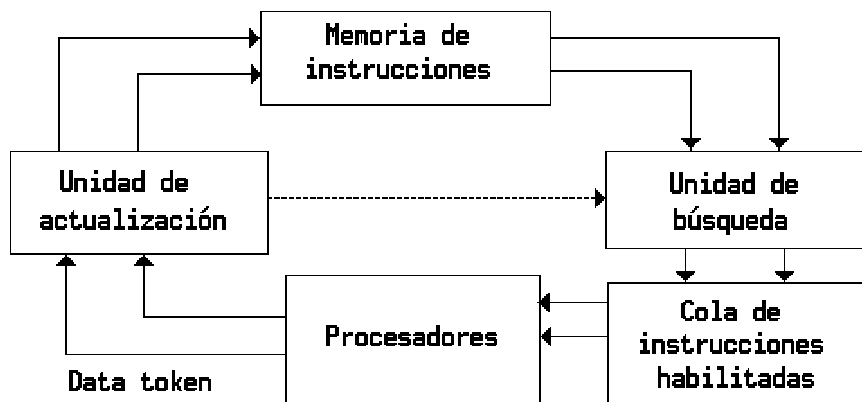


Fig. 9.9. - Arquitectura Dataflow Estática.

Las implementaciones estáticas cargan todos los nodos del grafo en memoria durante la inicialización y solo permiten que una instancia de un nodo se ejecute por vez.

En la Fig. 9.9 podemos visualizar la forma de esta arquitectura.

9.1.7.2 - **Arquitecturas Dinámicas**

En esta arquitectura se utilizan tokens identificados por medio de un rótulo (tags) de modo de permitir que más de un data token pueda estar sobre un arco, permitiendo de esta manera un mayor grado de paralelismo.

Una instrucción se ejecuta cuando tenga todos

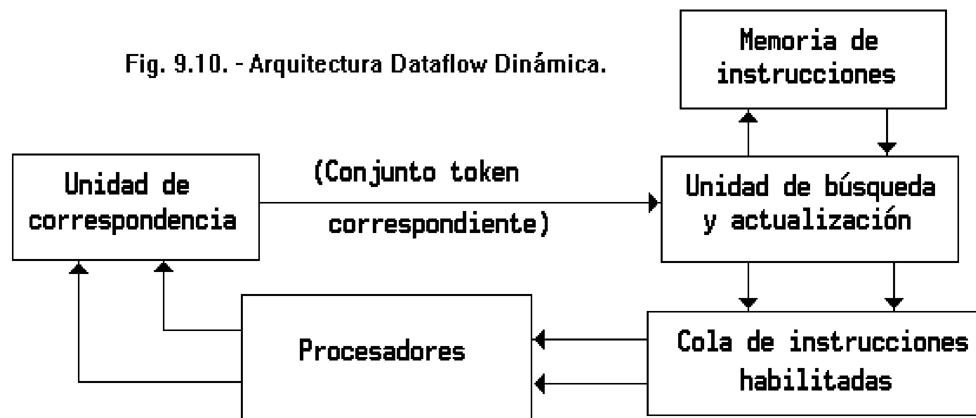


Fig. 9.10. - Arquitectura Dataflow Dinámica.

los data tokens presentes de un mismo tag (nivel).

Las implementaciones dinámicas permiten la creación de instancias de nodos en tiempo de ejecución y además que múltiples instancias de un nodo se puedan ejecutar concurrentemente.

Un ejemplo concreto de esta arquitectura es la Máquina de Manchester.

En la Fig. 9.10 esquematizamos esta arquitectura.

9.1.8 - Principio de asignación única

El modelo dataflow está estrechamente asociado al principio de asignación única (SA single assignment) lo que nos lleva al diseño de las arquitecturas de tipo DF.

El concepto esencial en el principio SA es la variable y cómo ésta se asigna en el programa.

En las computadoras tradicionales una posición de memoria está asignada a cada variable. Pueden asignarse diferentes valores de la misma variable durante el cálculo, entonces varios valores de la misma variable pueden aparecer en memoria en diferentes momentos. Por lo tanto, cuando una variable es accedida por una instrucción el momento del acceso adquiere significación así como el valor del identificador de la variable.

En la aplicación del principio de asignación única vale la siguiente regla para una variable :

Una variable puede tener un único valor durante el cálculo (y en un cálculo iterativo lo mantiene para un nivel de iteración).

Para el almacenamiento de una variable en el programa existen dos períodos :

- en el primer período se reserva una ubicación para la variable en memoria, pero no contiene ningún valor ; luego su valor no puede ser leído desde la memoria pero sí puede ser escrito,
- en el segundo período la variable ha adquirido su valor y ha sido grabada en la ubicación de memoria reservada; puede ser leída un número arbitrario de veces pero no puede ser modificada nunca más (consiguientemente no se puede realizar ninguna entrada en esa ubicación de memoria).

9.1.9 - Programación Dataflow

Los lenguajes de Data Flow pueden tener forma gráfica o léxica. Pueden estar a la altura de lenguajes de alto nivel, pero también pueden estar a la altura de lenguajes máquina.

Estos lenguajes tienen algunas propiedades dignas de atención :

- la escritura del programa es clara y de la misma forma es posible escribir un programa de "alto nivel" así como uno de "bajo nivel" ;
- los programas DF expresan de una manera simple el paralelismo natural de los algoritmos permitiendo la explotación del paralelismo a diferentes niveles ;
- parece posible trasladar en forma sencilla desde lenguajes de alto nivel programas ya hechos a lenguajes DF. Esto permite la utilización de programas existentes y hacer uso de los resultados que fueron obtenidos mediante la programación convencional en la programación DF.

Uno de los problemas más importantes con el que hay que vérselas y que tiene una gran influencia en la utilización del paralelismo, es las estructuras de los datos en los lenguajes de programación DF.

9.1.10 - Modelos básicos de sistemas Data Flow

Miller y Cocke (1972-1974) diseñaron dos modelos de sistemas que se han dado en llamar :

- Computador configurable de Modalidad Búsqueda (SM search mode).
- Computador configurable de Modalidad Interconexión (IM interconnection mode).

Estos modelos pueden ser considerados como los modelos básicos de la arquitectura de computadores DF.

Es posible considerar estos modelos básicos como casos extremos y en implementaciones particulares crear sistemas híbridos utilizando los principios de SM e IM en diferentes niveles de jerarquías de un sistema computador.

La mayor propiedad característica de ambos modelos es la posibilidad de la reconfiguración dinámica de la estructura del computador para procesar la tarea tan rápido como sea posible.

La computadora adapta dinámicamente su configuración a la estructura del algoritmo. Esto se logra interconectando (según grafo) los procesadores que se corresponden a operadores en el programa dataflow del problema.

En el tipo IM , la interconexión de los procesadores se implementa a través de un gran circuito (es decir vía hardware).

En el tipo SM la interconexión de los procesadores se simula utilizando un formato de instrucción especial (es decir vía software).

9.1.11 - La arquitectura Data Flow de Modalidad Búsqueda

La computadora de modalidad Búsqueda (Fig. 9.11) consiste de una memoria, una unidad funcional y una unidad de control que se ha dado en llamar "un buscador" (searcher).

La unidad funcional está compuesta por un cierto número de procesadores (por ejemplo : una red de microprocesadores).

Un buscador es una unidad especializada en generar tareas para los procesadores que pertenecen a la unidad funcional.

La memoria puede tener diversas estructuras y frecuentemente consiste de un conjunto de diferentes bloques especializados. Existen dos tipos posibles de memorias :

- memoria donde se almacenan datos e instrucciones
- memoria con almacenamiento separado para datos e instrucciones

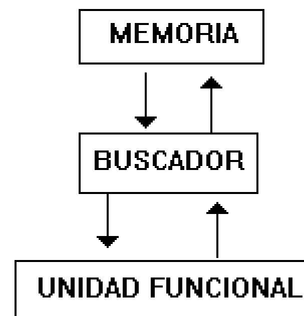


Fig. 9.11. - Computador configurable modalidad búsqueda.

Un procesador que esté libre consulta al buscador para que le dé una tarea. El buscador encuentra una tarea apropiada en memoria o compone una a partir de varios elementos almacenados en diferentes partes de la memoria y se la entrega al procesador seleccionado en la unidad funcional para su ejecución.

Pueden servir como procesadores :

- sumadores,
- multiplicadores,
- testers de errores,
- módulos especializados en macroperaciones,
- procesadores booleanos,
- procesadores de E/S,
- microprocesadores universales, etc.

La performance de un computador dependerá de la cantidad y tipo de procesadores que tenga, de la velocidad de la memoria, pero primordialmente del rendimiento del buscador del cual depende la efectiva utilización de la totalidad de sus procesadores.

El buscador realiza más de la mitad del trabajo que se requiere en un computador tradicional para la ejecución de una instrucción.

Para una descripción más detallada veamos el formato de una instrucción típica, en este caso una instrucción de dos operandos (Fig. 9.12).

Nombre	Operando_1	Operando_2	Dirección_del_resultado

Fig. 9.12. - Formato de instrucción en un dataflow SM.

El nombre y la dirección del resultado constituyen la parte de control de la instrucción, en tanto que los operandos constituyen la parte de datos de la instrucción.

El nombre contiene el código de operación, el nombre del programa y otros datos necesarios para sincronización, información de estado del cálculo, señales, etc.

Los campos de operandos contienen los valores de los operandos, aunque también pueden tener las direcciones de los mismos en memoria, y la dirección del resultado es la dirección donde debe entregarse el resultado luego de realizado el cálculo (podría ser la dirección en memoria, en alguna memoria auxiliar, o la dirección del próximo procesador, etc.).

Es decir, de esta forma, la instrucción tiene todos los datos necesarios para ejecutarse.

La instrucción puede almacenarse de esta forma en la memoria o puede ser generada así en el buscador. Este mecanismo asegura la "interconexión lógica" de los procesadores individuales en el programa y la continuidad del mismo.

Luego de que un procesador ejecutó una instrucción solicitará la siguiente al buscador; por lo tanto la velocidad del procesador depende de la disponibilidad de los datos y del flujo de datos del algoritmo.

Cada instrucción puede procesarse en el momento en que se encuentren disponibles sus operandos, pero si un procesador no está libre en ese momento, la sincronización del cómputo no se pierde. Más aún, las instrucciones pueden ejecutarse en un orden arbitrario. Esta propiedad es muy prometedora cuando se está diseñando una computadora paralela, ya que permite la explotación del paralelismo natural de las tareas así como una buena utilización de la memoria, ya que ambos, tanto las instrucciones como los datos, pueden estar distribuidos arbitrariamente en la memoria.

Cuál es la conexión entre los términos "token" e "instrucción ejecutable" (EI) ? Un token se utiliza principalmente para representar la dinámica del cálculo en el programa DF que se escribe en forma de grafo ; en tanto que una instrucción ejecutable se utiliza al describir la actividad en la computadora DF.

Por ejemplo en la Fig. 9.13 vemos la ejecución de una función f sobre dos operandos a y b como se la representa en el programa, y por otro lado se muestra cómo la ejecución de una función nos lleva a la generación de una EI en el computador.

Nótese que la EI es equivalente al token cuando el token está dentro del operador, es decir, cuando el operador se encuentra en estado activo, y también EI es equivalente al conjunto de tokens a y b cuando estos se encuentran en el input del operador, o sea, cuando este se encuentra en estado habilitado.

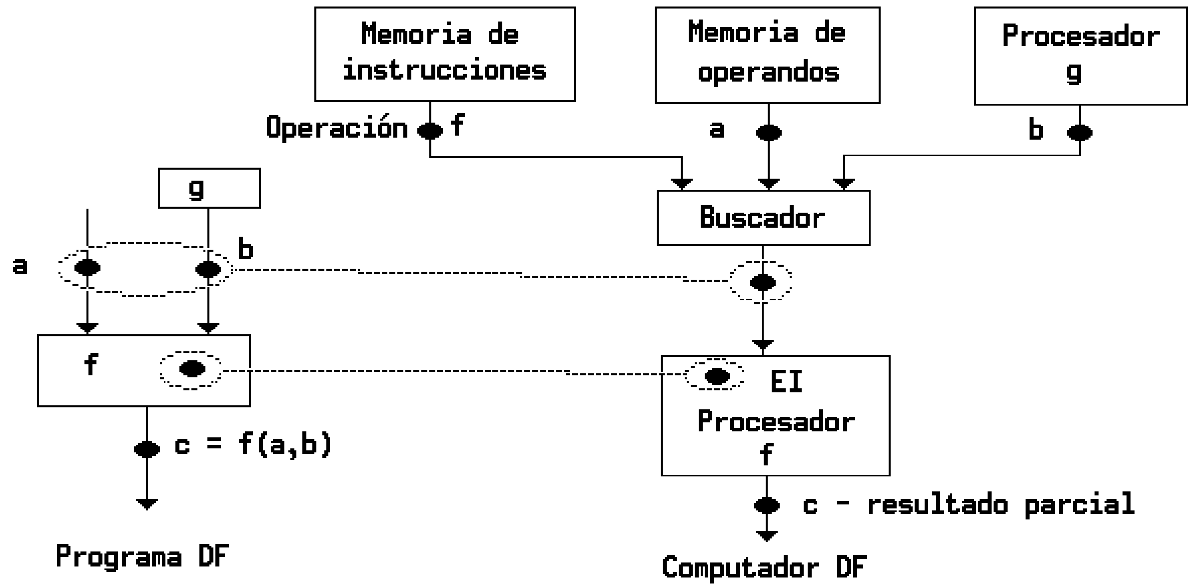


Fig. 9.13. - Token versus Instrucción Ejecutable (EI).

9.1.12 - **La arquitectura Data Flow de Modalidad Interconexión**

Al igual que en las computadoras SM las computadoras IM son también reconfigurables. En las IM la reconfiguración es más rígida.

El elemento característico del computador IM es un circuito o red de interconexión que ejecuta esta función de reconfiguración.

Este circuito provee una conexión directa entre los outputs de un grupo de procesadores y los inputs del próximo grupo de procesadores y crea de esta forma una red de procesadores correspondiente al programa DF que está siendo ejecutado.

Como regla, no es suficiente un conjunto de procesadores para crear una estructura de cómputo que pueda ejecutar el programa entero de una sola vez.

Entonces se hace necesario dividir el programa en bloques de un tamaño apropiado con respecto a la cantidad de procesadores disponibles por vez.

Esto se realiza a través de un compilador que determina la interconexión de los procesadores de manera tal que la estructura de cómputo corresponda al grafo del programa DF o a alguna parte del mismo. Esta interconexión se codifica y se almacena en memoria como una instrucción de seteo del circuito. Esta instrucción es accedida en primer término y llevada al control de seteo del circuito el cual realiza la interconexión de los procesadores. Así la estructura de cómputo está lista para poder comenzar a ejecutar el bloque de programa correspondiente.

Los operandos y los resultados parciales del cómputo de los bloques precedentes se toman de la memoria a través del control de acceso de datos antes de comenzar el cálculo.

Durante la ejecución del bloque no es necesario obtener ninguna instrucción ni datos desde la memoria, lo cual reduce la carga sobre los caminos de datos y sobre la memoria.

Luego de ejecutar un bloque los procesadores involucrados y la parte del circuito pertinente se liberan para poder ser nuevamente seteadas para otro bloque.

De esta manera la computadora IM aparece como un computador especializado mientras se encuentra ejecutando un bloque.

Resulta claro que otro bloque puede ser seteado antes de la finalización de la ejecución del precedente, y que los procesadores individuales pueden liberarse tan pronto como hayan terminado

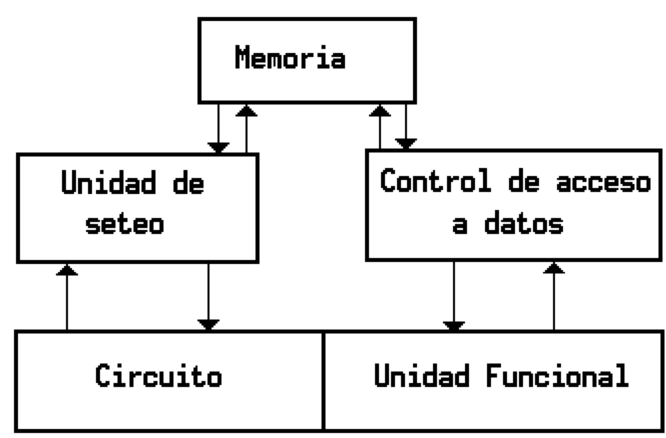


Fig. 9.14. - Computador configurable de modalidad interconexión.

su actividad en el bloque relevante, lo que puede llegar a ocurrir mucho antes de que el cálculo completo del bloque en su totalidad haya finalizado.

En la Fig. 9.14 podemos ver un esquema típico de la estructura de estos computadores.

9.1.13 - **Un mecanismo para implementar computadoras Dataflow**

Algunas arquitecturas almacenan directamente los tokens que contienen los resultados de las instrucciones en una plantilla o modelo (template) de la instrucción que los utilizará como operandos.

Otras arquitecturas utilizan esquemas de apareo de tokens (token matching), en las cuales una unidad de apareo almacena tokens y trata de hacerlos concordar o aparear con las instrucciones.

Cuando el conjunto completo de tokens (es decir todos los operandos) se reúnen para una instrucción, se crea y encola para su ejecución una plantilla que contiene los operandos relevantes.

La Fig. 9.15 muestra como una arquitectura simplificada de apareo de tokens puede procesar el fragmento de programa que se ve en la Fig. 9.16.

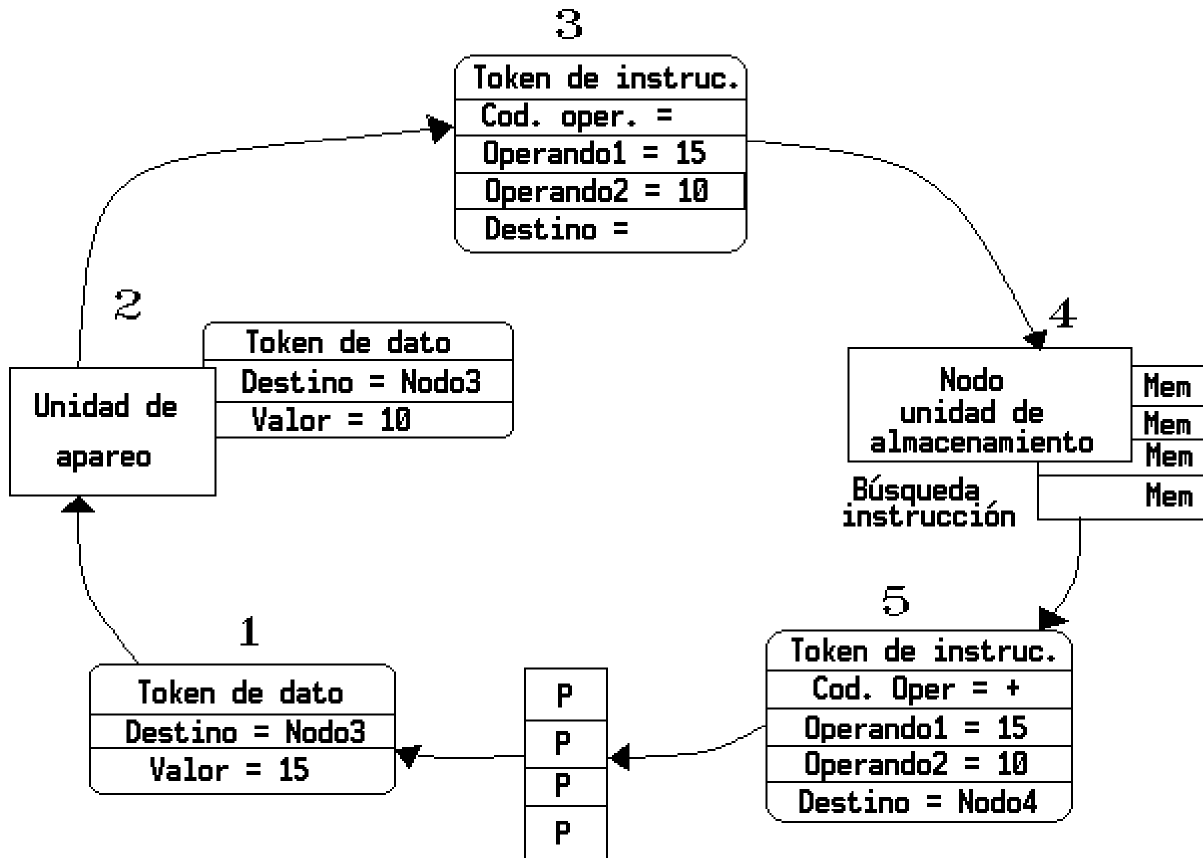


Fig. 9.15. - Ejemplo de apareo de tokens.

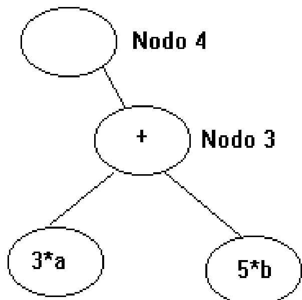


Fig. 9.16. - Fragmento de un grafo de un programa Dataflow.

En el paso 1, la ejecución de (3*a) resulta en la creación de un token que contiene el resultado (15) y una indicación de que la instrucción en el nodo 3 necesita de él como operando.

El paso 2 muestra como la unidad de apareo hace coincidir este token con el otro token que es resultado de realizar (5*b) y con la instrucción del nodo 3.

Esta unidad de apareo crea el token instrucción (la plantilla) que se ve en el paso 3.

En el paso 4 el nodo que hace de unidad de almacenamiento obtiene el código de operación relevante desde la memoria. Llena entonces los campos apropiados del token (paso 5) y asigna la ejecución a un procesador.

La ejecución de la instrucción creará un nuevo token resultado que se usará de input en el nodo 4.

9.2. - **ARQUITECTURAS MIMD/SIMD**

Durante los años 80 se construyeron una cantidad experimental de arquitecturas híbridas que permitían que partes de una arquitectura MIMD se controlaran en una modalidad SIMD (por ejemplo la DADO, la Non-Von, la Pasm y el Computador Array Reconfigurable de Texas o TRAC).

Los mecanismos explorados para reconfigurar arquitecturas y controlar la ejecución SIMD fueron asaz diversos. Usaremos como ejemplo de ayuda para poder ilustrar este concepto un sistema computador de pasaje de mensajes estructurado en forma de árbol (Ver Fig. 9.17).

La relación maestro/esclavo del controlador de la arquitectura SIMD y los procesadores puede mapearse hacia los nodos descendentes del subárbol.

Cuando el procesador raíz de un subárbol opera como un controlador SIMD se encarga de transmitir instrucciones a los nodos descendientes que, a su vez, las ejecutan en sus memorias locales.

La flexibilidad de las arquitecturas MIMD/SIMD las convierte en candidatas muy atractivas para que se continúe la investigación sobre ellas.

De entre los incentivos más recientes para invertir sobre nuevos desarrollos se cuentan el procesamiento paralelo de imágenes y las aplicaciones de sistemas expertos.

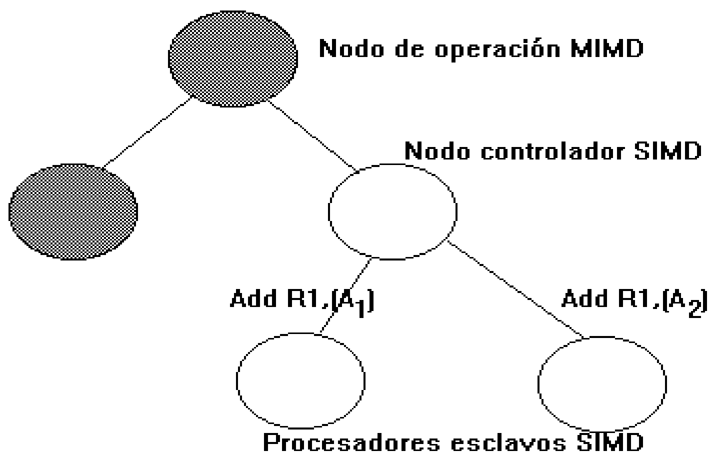


Fig. 9.17. - Operación MIMD/SIMD.

9.3 - ARQUITECTURAS DE REDUCCION

Las arquitecturas de Reducción o Demand-Driven (que se comportan según los requerimientos) implementan un paradigma de ejecución en el cual una instrucción está habilitada para realizarse cuando sus resultados son requeridos como operandos de otra instrucción que ya estuviera habilitada para ejecución.

Las investigaciones sobre estos tipos de arquitecturas comenzaron en los últimos años de la década del 70 con la finalidad de buscar nuevos ejemplos de ejecución paralela y para dar soporte a lenguajes de programación funcional.

Las arquitecturas de reducción ejecutan programas que consisten en expresiones anidadas. Las expresiones se definen recursivamente como literales o funciones de aplicación sobre argumentos que pueden ser literales o expresiones.

Los programas pueden "referenciarse" a expresiones por su nombre, las cuales devuelven siempre el mismo valor. Esta propiedad se denomina **propiedad de transparencia referencial**.

Los programas de reducción son aplicaciones de funciones construidos sobre funciones primitivas

La ejecución de un programa de reducción consiste en el reconocimiento de expresiones reducibles, y en reemplazarlas luego por sus valores calculados. Por lo tanto, el programa de reducción completo queda finalmente reducido a su resultado.

Debido a que el método general de ejecución solamente permite que una instrucción se ejecute cuando sus resultados son necesarios para una instrucción previamente habilitada, se requiere lógicamente de alguna regla adicional para habilitar la primera instrucción (o instrucciones) y poder comenzar el cálculo.

Existen objeciones prácticas para implementar las arquitecturas de reducción. Por ejemplo, la sincronización de los requerimientos para los resultados de una instrucción (ya que preservar la transparencia referencial exige que el cálculo de una expresión se realice solo una vez) y el mantenimiento de copias de los resultados de evaluación de las expresiones (ya que un resultado puede ser referenciado más de una vez pero puede ser consumido en las subsiguientes reducciones una vez que la primera se ha disparado).

Las arquitecturas de reducción utilizan tanto reducción de strings como reducción de grafos.

La reducción de strings implica manejar literales y copias de valores, que se representan como strings que pueden expandirse o contraerse dinámicamente.

La reducción de grafos implica manejar literales y referencias (pointers) a valores.

Luego, un programa se representa como un grafo, y un método de "garbage collection" obtiene dinámicamente memoria a medida que la reducción avanza.

Las Fig. 9.18 y 9.19 muestran una versión simplificada de una arquitectura de reducción de grafos que mapea el programa que sigue en una estructura arbórea de procesadores y que pasa tokens a requerimiento y devuelve resultados.

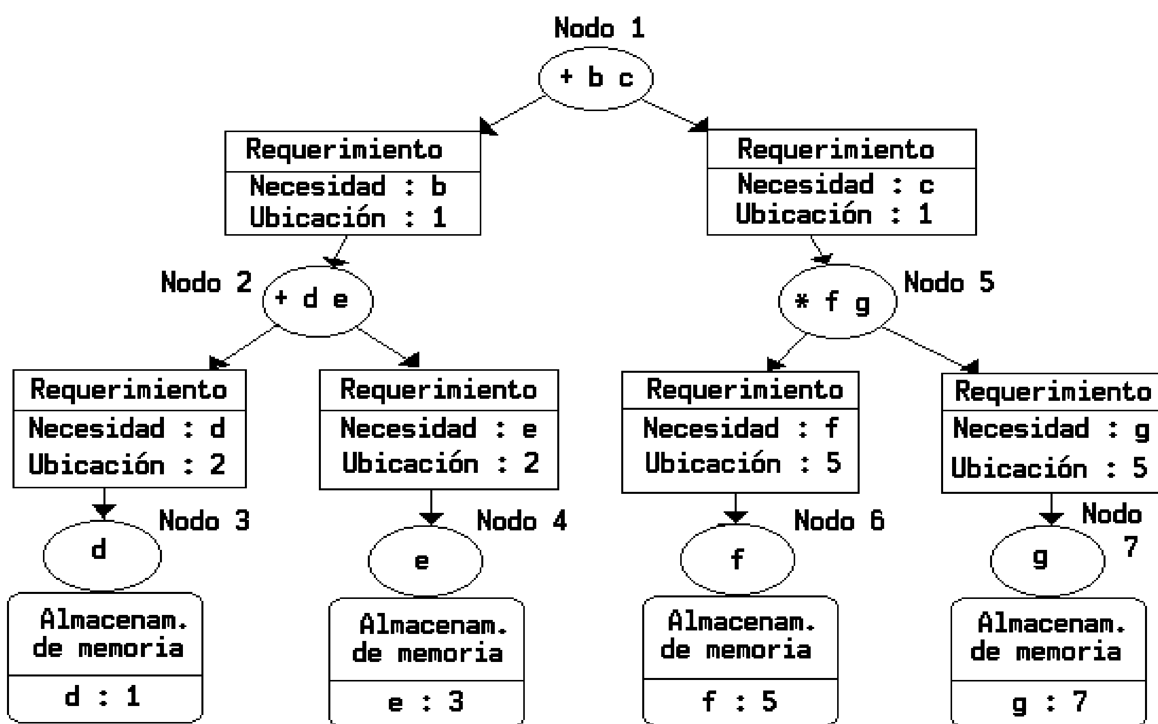


Fig. 9.18. - Arquitectura de Reducción, tokens producidos a requerimiento del programa.

a = + b c ;
b = + d e ;
c = * f g ;
d = 1;
e = 3;
f = 5;
g = 7.

La Fig. 9.18 muestra todos los tokens producidos a requerimiento del programa, así como su propagación hacia abajo en el árbol. En la Fig. 9.19 se muestra como los dos últimos tokens de resultados producidos son pasados al nodo raíz.

Se han propuesto arquitecturas muy disimilares para soportar reducción de string y de grafos, por ejemplo la Máquina de Reducción Newcastle, la Máquina North Carolina Cellular Tree, y el Sistema Utah Multiprocesamiento Aplicativo.

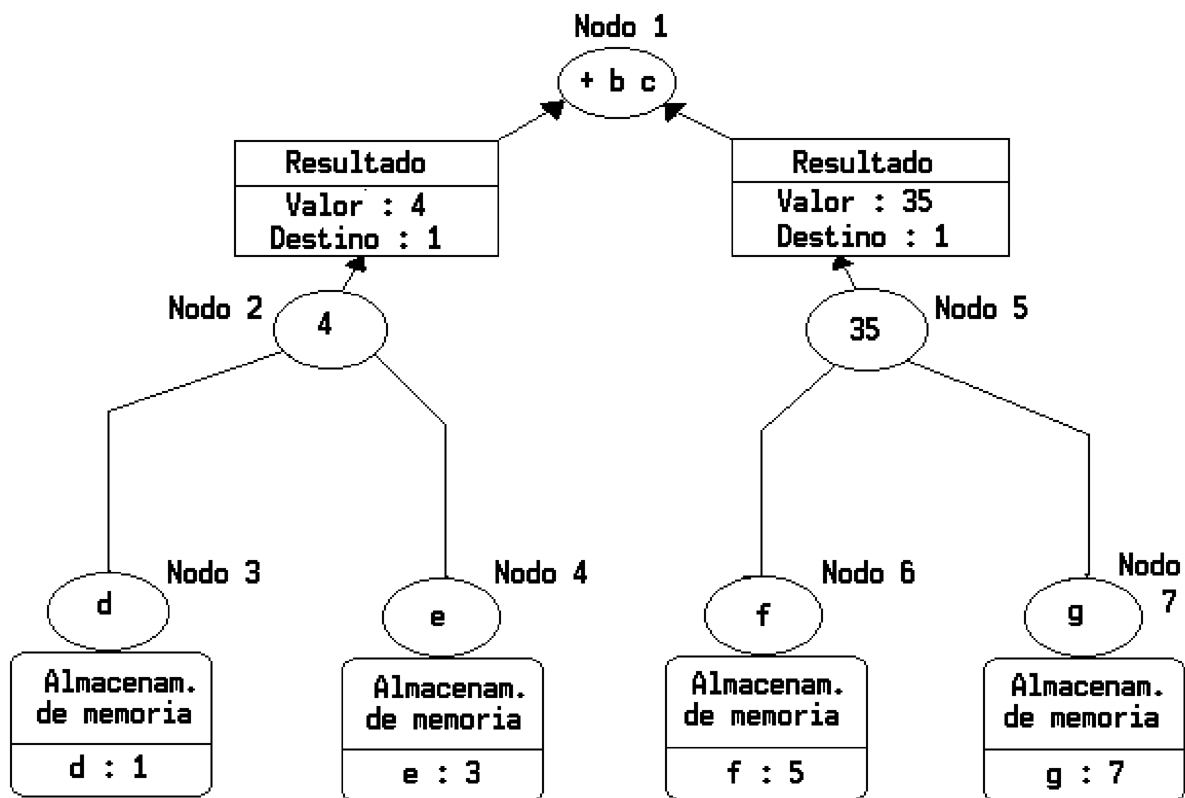


Fig. 9.19. - Arquitectura de Reducción, tokens de resultado.

9.4 - **ARQUITECTURAS WAVEFRONT ARRAY**

Los Wavefront array combinan el pipelining de datos de los sistólicos con el flujo asincrónico de la ejecución de las Dataflow.
S. Y. Kung desarrolló los conceptos de las Wavefront a fines de los años 80 para resolver el mismo tipo de problemas que estimuló el desarrollo de los sistólicos : producir arquitecturas eficientes de un costo aceptable para sistemas de propósito específico que balancearan una gran cantidad de cálculos con un amplio bandwidth de E/S.

Ambos, Wavefront y Sistólicos, se caracterizan por poseer procesadores modulares y regulares, y una red de interconexión local.

Sin embargo los wavefront reemplazan el reloj global y los mecanismos explícitos de retardo para sincronizar el pipelining de los datos por un mecanismo de handshaking (acuerdo) asincrónico para coordinar los movimientos de los datos entre los procesadores.

Luego, cuando un procesador ha realizado su cálculo y está listo para pasar los datos a su sucesor, informa al sucesor y cuando éste le dice que se encuentra listo mediante la señal de acknowledge, le envía los datos.

El mecanismo de handshaking hace que el cálculo en el wavefront se realice fácilmente a través del array sin que ocurran intersecciones (o colisiones) y de esta forma el procesador array se comporta como un medio de propagación de olas (wave).

De esta forma un secuenciamiento correcto del cálculo reemplaza al correcto sincronismo de las arquitecturas sistólicas.

En las Fig. 9.20, 9.21 y 9.22 se muestran los conceptos del array wavefront utilizando un ejemplo de multiplicación de dos matrices de 2 * 2 que son de la siguiente forma :

A = a c B = e g
b d f h

La arquitectura de ejemplo utiliza elementos de procesamiento (PE) que tienen un buffer de un operando para cada punto de entrada. Toda vez que el buffer de entrada está vacío y la memoria asociada a ese buffer contiene un nuevo operando, entonces el operando es inmediatamente leído. Los operandos de los otros PEs se obtienen usando el protocolo de handshaking.

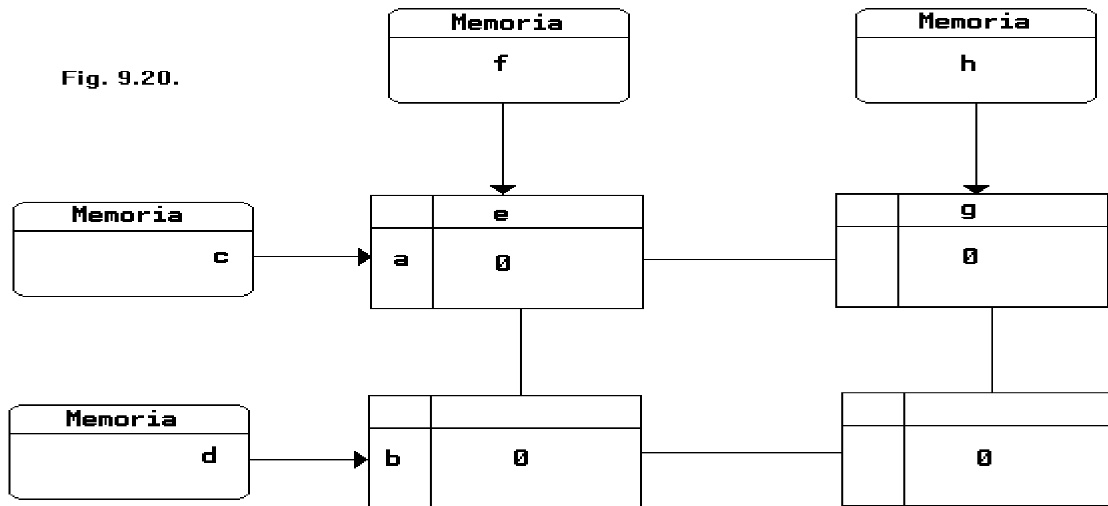


Fig. 9.20.

La Fig. 9.20 muestra la situación inicial cuando se encuentran llenos todos los buffers de entrada.

En la Fig. 9.21 el PE(1,1) suma el producto **ae** a su acumulador y transmite el operando **a** y el **e** a sus veci-

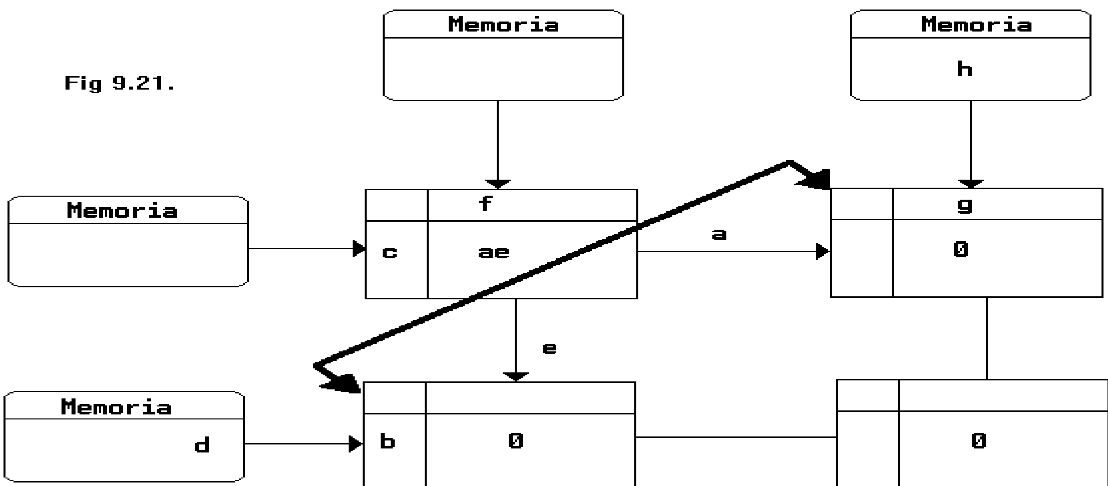


Fig 9.21.

nos; entonces puede verse el primer cálculo wavefront propagándose desde el PE(1,1) al PE(1,2) y al PE(2,1).

La Fig. 9.22 muestra como el primer cálculo continúa propagándose mientras que la segunda ola se empieza a propagar desde el PE(1,1).

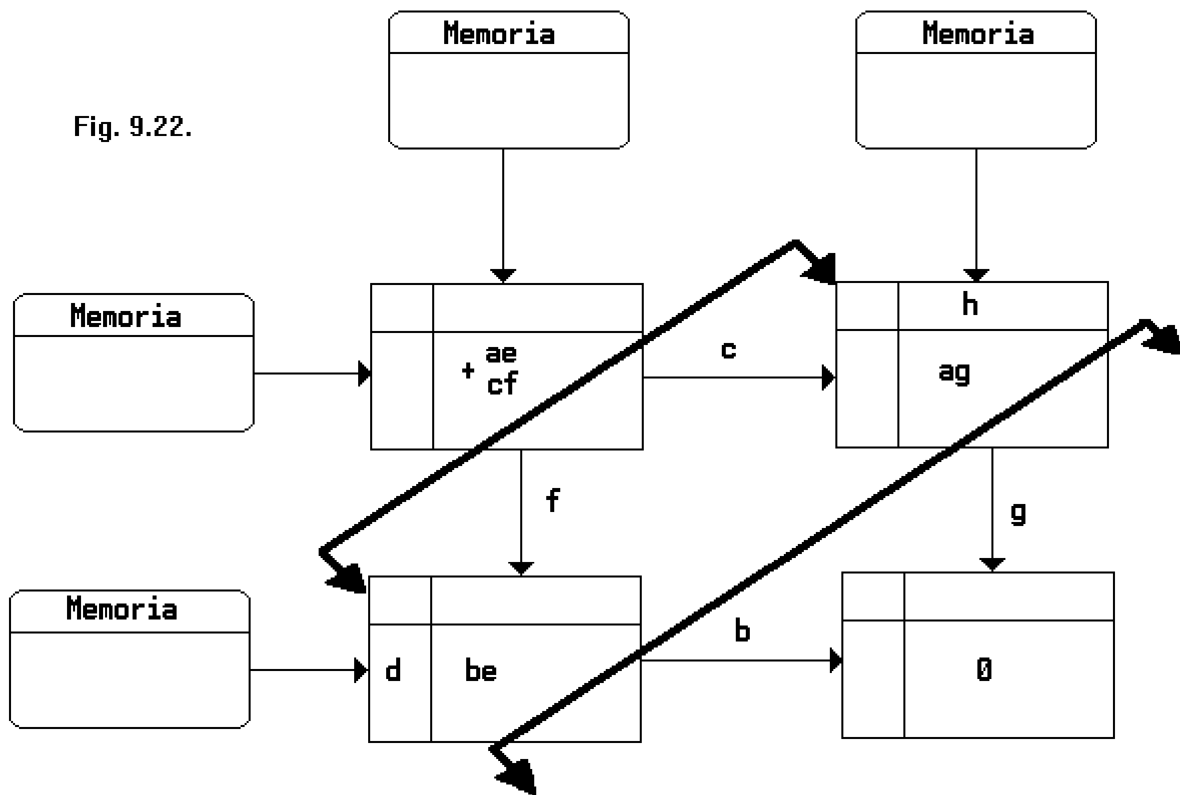


Fig. 9.22.

Kung argumenta que los wavefront gozan de diferentes ventajas respecto de los array sistólicos, como por ejemplo, que son más escalables (pueden crecer más fácilmente), son más simples de programar y son más tolerantes a fallas.

EJERCICIOS

1) Cómo es el modelo de cómputo Dataflow y en qué se diferencia del modelo de cómputo de Von Neumann ? Cuál es el elemento clave que los diferencia exactamente ?
2) Cuáles son los tipos fundamentales de operadores en el modelo Dataflow ?
3) Qué es un Token ? Qué clase de información puede contener ?
4) Qué significa que un operador se encuentra en estado "activo" ?
5) Cuál es la importancia de los intérpretes no-feedback ?
6) Qué es una arquitectura dataflow estática ?
7) Qué tipo de intérprete se utilizará en una arquitectura dataflow dinámica ? Justifique.
8) Cuál es el principio de asignación única ?
9) Cómo es la arquitectura Dataflow de modalidad búsqueda ?
10) En qué situación una instrucción ejecutable es equivalente a un token en una máquina Dataflow ? Justifique.
11) Cuál es la diferencia entre una arquitectura Dataflow de modalidad búsqueda y una de modalidad interconexión ?
12) En qué se basan las arquitecturas híbridas de tipo MIMD/SIMD ?
13) En qué principio se basan las arquitecturas de Reducción ?
14) Qué significa la propiedad de transparencia referencial ?
15) Cuáles son las dos características fundamentales de las arquitecturas Wavefront array ?

CAPITULO 10

TAXONOMIAS DE ARQUITECTURAS DE COMPUTADORAS

10.1. - **Introducción**

La clasificación de Flynn no discrimina claramente las diferentes arquitecturas de multiprocesadores. Ya que se ha incrementado substancialmente la cantidad de arquitecturas de multiprocesadores, se ha vuelto más importante encontrar una manera útil de describirlas (una forma que distinga aquellas que son significativamente diferentes en tanto que muestre las similitudes subyacentes entre los diseños aparentemente divergentes).

Este capítulo presenta una taxonomía (Skillicorn 1988) para las arquitecturas de computadores que extiende la de Flynn, especialmente en lo que respecta a la categoría de los multiprocesadores.

Esta clasificación no se opone a lo expresado en el capítulo 6 sino que se utiliza, en esta oportunidad, un mecanismo de definición más formal.

Esta taxonomía es una jerarquía de dos niveles, en la cual el nivel superior clasifica las arquitecturas basadas en la cantidad de procesadores para datos y para instrucciones y las interconexiones entre ellos.

El nivel inferior que puede usarse para diferenciar con más precisión las variantes, está basado en una visión de los procesadores como máquinas de estado.

Existe una única taxonomía formal, debida a Flynn, que se usa en general y no alcanza a diferenciar la inmensa variedad de arquitecturas posibles de multiprocesadores. Flynn clasifica las arquitecturas por el número de conjuntos de instrucción y conjuntos de datos que pueden procesar simultáneamente.

La taxonomía que aquí se presenta está basada en una visión funcional de la arquitectura y en el flujo de información entre las unidades.

Esta taxonomía es una estructura de dos niveles que subsume la clasificación de Flynn. Las discriminaciones de grueso calibre pueden realizarse en el nivel superior, en tanto que las distinciones más finas se hacen en el segundo nivel.

Entre otras clasificaciones que hay en la literatura, es la clasificación de Feng (1972), quizás la más conocida. Es una clasificación orientada a performance que describe el paralelismo de un conjunto de procesadores en una máquina en términos de la cantidad de bits que pueden procesarse simultáneamente.

Las máquinas se describen por un par ordenado, el primer elemento indica el tamaño de la palabra, y el segundo la profundidad (la cantidad de palabras que pueden operar simultáneamente). Esto permite comparaciones de performance entre una amplia variedad de arquitecturas, pero, justamente, como relaciona arquitecturas diversas no hace resaltar sus diferencias.

Otra clasificación debida a Reddi y Feurstel (1976) utiliza la organización física, el flujo de información, y la representación y transformación de la información como base para la clasificación. El flujo de la información puede ser una herramienta muy poderosa y general para describir arquitecturas, pero sus otros atributos dependen en demasía del tipo de implementación concreta.

Otra clasificación muy popular es la de Händler (1977), que describe las arquitecturas mediante la cantidad de procesadores y la forma en que pueden ser pipelinizados, y el tamaño y profundidad de las unidades aritmético-lógicas.

En tanto que estas clasificaciones son buenas para las arquitecturas vectorizadas convencionales, no aportan claridad respecto de generalizar sobre las nuevas arquitecturas de multiprocesadores.

10.2. - **Razones para un modelo arquitectural**

Existen tres razones para clasificar arquitecturas. La primera es comprender el punto que se ha alcanzado actualmente. Hasta las pasadas dos décadas, casi todos los sistemas de computadoras usaron la arquitectura Von Neumann. Incluso cuando el hardware subyacente comenzó a tener cierta capacidad limitada de paralelismo (como el caso del CDC6600), este se ocultó generalmente al usuario. Sin embargo, a partir de entonces, el crecimiento en sistemas con diferentes clases de paralelismo fue explosivo, y no está del todo claro cuáles de estas arquitecturas tienen las mejores proyecciones hacia el futuro.

La segunda razón para tener una clasificación de arquitecturas es el hecho de que la misma puede revelar configuraciones posibles que podrían no habérsele ocurrido jamás a un diseñador. Una vez que los sistemas existentes se hubieran clasificado, las lagunas en la clasificación pueden sugerir otras posibilidades. Por supuesto que no todas estas posibilidades pueden implicar mejoras, bien pudieron haber sido probadas y descartadas por peores.

Sin embargo, hasta que el espacio de búsqueda haya sido claramente delimitado, no puede estarse seguro de que todas las posibilidades hayan sido examinadas.

La tercera razón para un esquema de clasificación es el hecho de que ella permite que se construyan y utilicen modelos más útiles de performance. Un buen esquema de clasificación podría revelar porqué una arquitectura en particular es mejor para proveer una determinada mejora de performance. Puede servir también como modelo para análisis de performance.

10.3. **Intento de clasificación**

Si consideramos la completa cuestión del cómputo y de las máquinas que lo proveen, parece útil pensar en ciertos niveles de abstracción.

En el más alto, el nivel abstracto, podemos considerar el modelo de cómputo que se está utilizando. Muchas computadoras aún emplean el modelo de cómputo denominado el modelo de Von Neumann.

Desde este punto de vista del cómputo, las operaciones primitivas consisten en las operaciones usuales de la matemática: suma, resta, multiplicación, comparación, y así siguiendo (en dominios convenientemente restringidos).

Los datos primitivos, sin embargo, permanecen en las denominadas posiciones (locations), y el comportamiento correcto del cómputo se cumplimenta con el programador que describe el orden exacto en el cual los cálculos deben realizarse.

De hecho, el orden de ejecución está superrestringido, en el sentido de que existen otras secuencias de ordenamiento triviales que calculan el mismo resultado, pero que no existe forma de expresarlas en el modelo de cómputo.

El modelo de cómputo de Von Neumann parece natural a aquellos cuya experiencia de programación fue desarrollada utilizando lenguajes que soportan este modelo. Sin embargo, simplemente no contiene ningún concepto de cómputo más que el de hacer una cosa a la vez (paralelismo), ni tampoco alguna forma mediante la cual sea posible expresar un ordenamiento del cómputo dinámicamente.

No es de sorprender que los primeros nuevos modelos de cómputo desarrollados fueran extensiones del modelo de Von Neumann, a los cuales se agregaran conceptos tales como la comunicación de primitivas. Luego se desarrollaron otros modelos más inusuales (tales como dataflow y reducción de grafos).

A medida que nos acercamos a un nivel más detallado, podemos considerar las **máquinas abstractas** como la implementación de los modelos de cómputo.

El mapeo desde el modelo de cómputo hacia la máquina abstracta no es necesariamente una relación uno-a-uno; algunos modelos de cómputo pueden implementarse en más de un tipo de máquina abstracta, aunque alguna de ellas se adapte mejor que las otras.

Una máquina abstracta captura la esencia de la forma de una arquitectura en particular, sin hacer distinción entre las diferentes tecnologías, tamaño de la implementación, o consideraciones semejantes.

Por el momento, la máquina abstracta tradicional de Von Neumann puede resumirse como un contador de programa, modos de direccionamiento, códigos de condición, una unidad de control, y una unidad aritmético y lógica.

Esta máquina abstracta no hace referencia a las diferentes implementaciones Von Neumann que existen, ni diferencia entre RISC y CISC. Este es sin embargo un nivel muy útil en el cual considerar las arquitecturas, y es el nivel que forma la clasificación más alta en esta taxonomía.

En el siguiente nivel, más detallado, consideramos las implementaciones de máquina. Esto no solo concierne a la tecnología utilizada para implementar la máquina. Más bien, este nivel representa una definición de la arquitectura del computador, la ilustración (o el dibujo) que se muestra al programador de lenguaje assembler. Este nivel corresponde al segundo de la taxonomía.

No tratamos aquí el papel que juegan los lenguajes de programación en esta clasificación, ya que, en cierto sentido el lenguaje utilizado en una máquina corresponde al modelo de cómputo que ella posee. Si no es este el caso, entonces existe un bastante mal ajuste entre el lenguaje y la implementación, y ya que nuestro objetivo es la performance, podemos ignorar esta clase de situaciones.

Por supuesto que cuando todas las máquinas que había eran Von Neumann, se utilizaron muchos diferentes lenguajes en la misma máquina. Sin embargo, ha existido una tendencia creciente hacia la construcción de la arquitectura y el lenguaje en forma conjunta, y existen muy buenas razones para tal tendencia. Aún en el caso de la arquitectura Von Neumann surge claramente que los lenguajes de tipo Algol se adaptan mejor que por ejemplo el Lisp.

10.4. - **Funciones de la arquitectura del computador**

En esta clasificación de las arquitecturas de los computadores utilizamos cuatro tipos de unidades funcionales a partir de las cuales se puede construir una máquina abstracta. Ellas son :

- Un procesador de instrucciones; una unidad funcional que actúa como un intérprete de las instrucciones, cuando tales elementos existen explícitamente en el modelo de cómputo.
- Un procesador de datos; una unidad funcional que actúa como un transformador de datos, normalmente de la forma que se corresponda a las operaciones aritméticas básicas.
- Una jerarquía de memoria; un dispositivo de almacenamiento inteligente que transmite datos de y hacia los procesadores.
- Un switch; un dispositivo abstracto que provee conectividad entre las otras unidades funcionales.

Veremos a continuación cómo estas unidades funcionales representan el comportamiento de una máquina abstracta utilizando la arquitectura de Von Neumann como un ejemplo.

10.5. - **LA MAQUINA ABSTRACTA VON NEUMANN**

La máquina abstracta Von Neumann consta de un solo procesador de instrucción (PI), un sólo procesador de datos (PD), y dos memorias jerárquicas. La unidad funcional switch (SW) no juega ningún papel en la máquina abstracta Von Neumann.

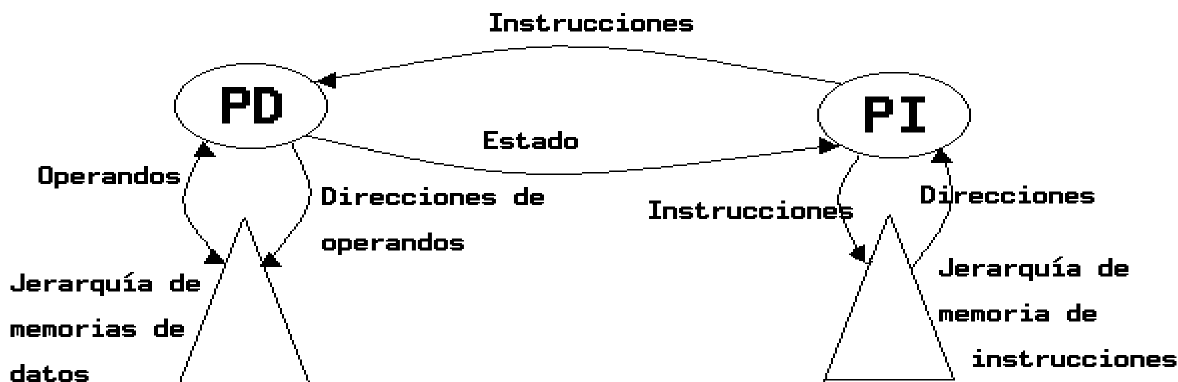


Fig. 10.1. - Disposición de las unidades funcionales de la máquina abstracta Von Neumann.

Estas unidades funcionales están dispuestas como muestra la Fig. 10.1. El procesador de instrucción está conectado a una de las jerarquías de memoria de donde toma las instrucciones. Para hacer esto él debe proveer e la memoria de un label o dirección que identifique la instrucción que él desea acceder.

Está conectado también al procesador de datos, al cual entrega información acerca de las operaciones a realizar y las direcciones de los valores sobre los cuales la operación debe realizarse, y recibe de éste la información de estado (códigos de condición).

Las funciones del procesador de instrucción son :

(1) Determinar la dirección de la próxima instrucción a ejecutar, utilizando la información de su propio almacenamiento interno y la información de estado que le fuera enviada desde el procesador de datos.
(2) Dar la instrucción a la memoria jerárquica para que esta le provea la instrucción con tal dirección.
(3) Recibir la instrucción y decodificarla. Esto implica determinar cuál operación, de existir, deberá ser requerida al procesador de datos para "ejecutar" la instrucción.
(4) Informar al procesador de datos sobre la operación requerida.
(5) Determinar las direcciones de los operandos y pasarlas al procesador de datos.
(6) Recibir la información de estado desde el procesador de datos (luego de completada la operación).

Cualquier arquitectura con un procesador de instrucciones requerirá una serie de pasos como los descriptos arriba.

No existe ningún supuesto en esta descripción sobre el hecho de que estos pasos se realicen en este orden o que alguno de ellos pueda saltearse.

En el primer nivel de la taxonomía se especifican solamente la existencia de ciertas unidades funcionales conectadas de una cierta forma.

En la máquina Von Neumann, el orden de las operaciones es normalmente el indicado antes. Este se denomina la secuencia de ejecución de la instrucción.

Si deseamos indicar precisamente el orden de los pasos debemos pasar al siguiente nivel de la taxonomía y representar estas diferentes operaciones como estados en un diagrama de estados.

En este segundo nivel el procesador de instrucciones puede representarse como se ve en la Fig. 10.2, en la cual cada estado representa una operación y las flechas indican las dependencias entre los estados. Las flechas

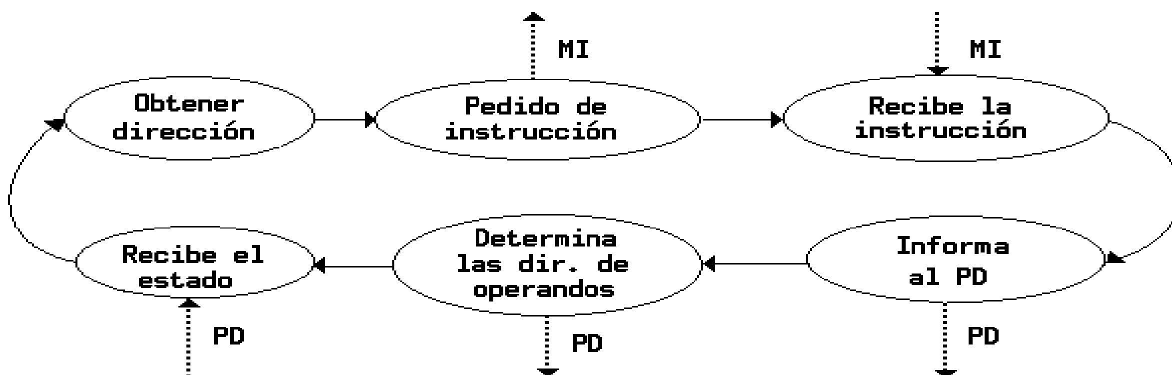


Fig. 10.2. - La estructura interna de un procesador de instrucción Von Neumann.

punteadas representan la comunicación entre el procesador de instrucción y las otras unidades funcionales del sistema.

Si podemos colocar un token para un determinado estado, entonces este diagrama de estados (conjuntamente con las ubicaciones de los tokens) da una completa descripción sobre qué está haciendo el procesador de instrucción en un momento dado. El token va siguiendo las flechas completando el circuito para cada instrucción ejecutada.

Podemos seguir haciendo diferenciaciones de acuerdo al diagrama de estados presentado indicando detalles a nivel de la implementación. Esto podría incluir mecanismos o controles temporales para cada estado, o la forma en que las entidades lógicas se mapean en entidades físicas.

Un ejemplo de esto es el siguiente: Muchos de los computadores Von Neumann implementan la selección de la próxima instrucción utilizando un contador de programa que se actualiza con la longitud de la instrucción actual (ignorando las instrucciones de bifurcación). Sin embargo, en algunos sistemas la dirección de la próxima instrucción está incluida en la instrucción en sí misma. Ninguna de estas estrategias es más Von Neumann que la otra; son elecciones de implementación y son indistintas a nivel de la máquina abstracta.

El procesador de datos lleva a cabo los siguientes pasos :

(1) Recibe un tipo de instrucción desde el procesador de instrucción.
(2) Recibe las direcciones de operandos desde el procesador de instrucción.
(3) Da las instrucciones a la memoria jerárquica (separada de la del procesador de instrucción) para que ésta le provea los valores de los operandos.
(4) Recibe los valores de los operandos desde la memoria jerárquica.
(5) Realiza la operación requerida (la fase de ejecución).
(6) Devuelve información de estado al procesador de instrucción.
(7) Provee un valor resultado a la memoria jerárquica.

Una vez más, esto puede representarse mediante un diagrama de estados consistente en un ciclo de estados.

La mayor diferencia con el procesador de instrucción es que la fase de ejecución del procesador de datos consiste potencialmente de una gran cantidad de estados paralelos que representan aquellas operaciones que el procesador de instrucción ve como una primitiva. El diagrama de estado se ve en la Fig. 10.3.

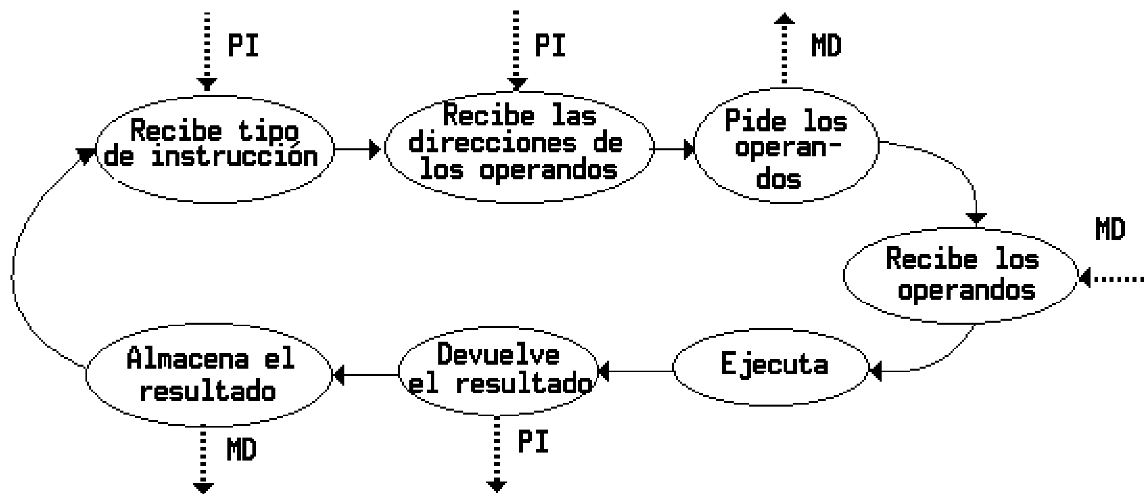


Fig. 10.3. - La estructura interna del procesador de datos de Von Neumann.

Ciertas acciones que se realizan en ambos procesadores deben ser sincrónicas. Los lugares donde esto ocurre se grafican utilizando líneas punteadas en el diagrama de estados, y representan la comunicación entre los procesadores.

Su comportamiento es el siguiente : debe existir un token al comienzo y en la finalización del camino de comunicación. Ningún token puede abandonar el estado hasta que la comunicación se haya completado (es decir, la comunicación es sincrónica).

La memoria jerárquica es un dispositivo inteligente que intenta proveer los datos que le requieren los procesadores unidos a ella, lo más rápido posible.

El sistema de almacenamiento puede ser visto como una pirámide, o más exactamente como un zigurat, ya que los datos almacenados son discretos y las capas sucesivas se amplían en etapas.

Una memoria jerárquica trata de tener la próxima porción de datos que le será requerida por el procesador que se comunica con el tope de la jerarquía donde el tiempo de acceso es el menor. Una jerarquía perfecta de memoria alcanzaría siempre este objetivo. Sin embargo, no existe una forma óptima de determinar qué porción de datos será la próxima requerida. De allí, que todas las jerarquías de memoria deben aproximar tal objetivo mediante heurística.

Lógicamente, el promedio de tiempo de acceso se minimiza cuando los datos más referenciados permanecen en la jerarquía más alta y aquellos referenciados menos a menudo permanecen en un nivel inferior.

Cuando los datos accedidos tienen algún tipo de localidad, ya sea temporal o espacial, la performance del tiempo promedio de acceso puede mejorarse manteniendo los datos que hayan sido recientemente accedidos o los que se encuentren cerca de lo accedido en la jerarquía más alta.

Podemos ver ahora porqué una máquina abstracta Von Neumann tiene dos jerarquías de memorias. Los datos en la memoria de instrucción fluyen hacia el procesador; las instrucciones son consumidas por el procesador de instrucciones y no se modifican a sí mismas. Por otra parte, la jerarquía de memoria de datos debe manejar el movimiento de los datos en ambas direcciones. De allí que la implementación de las dos jerarquías de memoria debe tener propiedades substancialmente diferentes.

Normalmente las jerarquías de memoria se implementan utilizando una sola jerarquía de almacenamiento excepto en el tope (la cache) en el cual se diferencian los datos e instrucciones. La cache está en el tope de la jerarquía de memorias, por encima de la memoria principal, la cual se continúa a su vez en el almacenamiento en discos y en términos de mayor capacidad hacia los dispositivos en cinta. Las E/S se producen en el nivel más bajo de la jerarquía de memorias. En un cierto sentido, la E/S es un acceso a un mecanismo de almacenamiento de gran capacidad, el mundo externo.

10.6. - **FORMAS DE INCREMENTAR PERFORMANCE.**

Muchos de los otros modelos de cómputo fueron motivados por el deseo de mejorar la performance, por tanto es instructivo en este punto considerar cómo la máquina abstracta Von Neumann puede rendir una mejor performance.

Existen tres grandes formas :

- reordenar el diagrama de estados a fin de que el token pueda circular en el menor tiempo;
- permitiendo que más de un token, y por tanto más de un paso activo, existan simultáneamente, y
- duplicando unidades funcionales para permitir la actividad concurrente.

Todas estas tres variantes incrementan la performance y en particular la tercera es muy fructífera y ha llevado a una amplia variedad de clases de arquitecturas.

El primer método para incrementar la velocidad (reordenando el diagrama de estados), no genera realmente una nueva clase de arquitectura, aún así el esquema de clasificación permite distinguir ciertas diferencias en la descripción de los estados del procesador.

Como un ejemplo, en el procesador de datos las operaciones de informar al procesador de instrucciones sobre el estado de la información derivada del resultado y el almacenamiento del resultado en las memorias jerárquicas son independientes. Sin embargo, pueden ocurrir simultáneamente. Esto nos lleva al diagrama revisado que puede verse en la Fig. 10.4, el cual nos muestra, desde el punto de vista del tiempo de cada etapa, que el nuevo ciclo será más veloz.

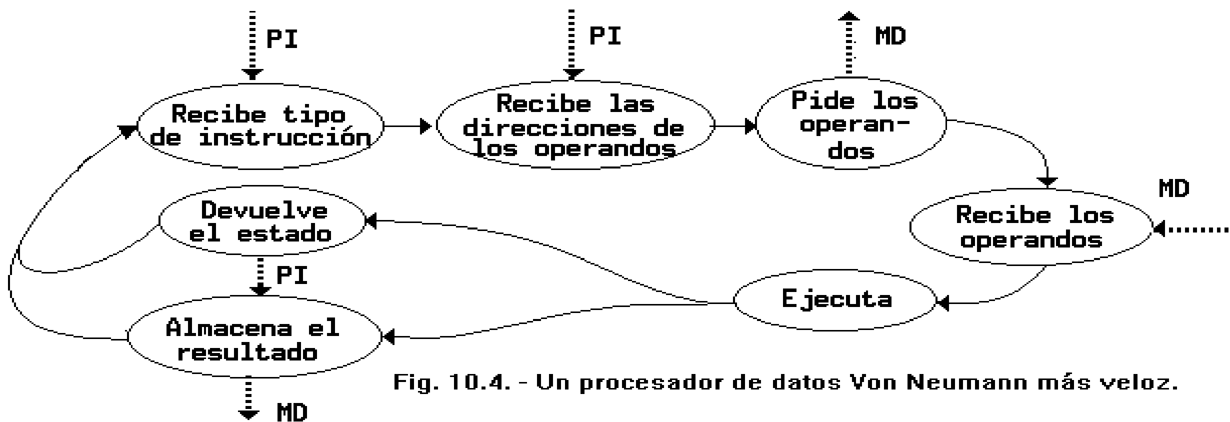


Fig. 10.4. - Un procesador de datos Von Neumann más veloz.

10.6.1. - **Pipelining**

La segunda posibilidad de incrementar velocidad permite que existan más de un token en el diagrama de estados, y se denomina Pipelining.

Supongamos que el número de estadios en el diagrama de estados es n. Si cada estadio demora la misma cantidad de tiempo, por ejemplo t, el tiempo de ejecutar una instrucción será n * t. Si permitimos que n tokens estén presentes simultáneamente, estamos representando n diferentes instrucciones en varios estadios de ejecución.

El tiempo que lleva ejecutar una instrucción sigue siendo n. Sin embargo, el tiempo promedio en el cual las instrucciones se completan es de 1/t instrucciones por unidad de tiempo.

Luego, hemos alcanzado un mecanismo acelerador de n-partes en nuestra máquina, de manera tal que tengamos siempre n tokens presentes en el diagrama de estados.

Por supuesto que existen dificultades prácticas para realizar esto mismo (instrucciones tales como bifurcaciones y saltos, interrupciones externas, y necesidades simultáneas de acceder a los mismos datos) que provocan que la performance real del pipeline sea mucho peor de lo que nuestro análisis sugiere. Sin embargo, sirve para

demostrar porqué el pipelining fue de interés a los diseñadores de sistemas que intentaban mejorar la performance de las máquinas Von Neumann.

Los pipelines tienen otra interesante propiedad. Si subdividimos cada uno de los estados en subestados, de manera que cada uno lleve un tiempo de t/2, y permitimos que existan 2n tokens en el diagrama de estados, entonces una instrucción se completa cada t/2 unidades de tiempo, incrementando el promedio de completitud de instrucciones a 2 / t instrucciones por unidad de tiempo. Luego, tenemos una aceleración total de 2n.

Haciendo más pequeño cada estadio nos permite incrementar aún más el promedio de finalización, pero a expensas de complicar el manejo de interacciones imprevistas.

Dentro de nuestro modelo de máquina abstracta el comportamiento pipeline puede describirse sin agregar ninguna unidad funcional nueva. En cambio, rotulamos cada unidad funcional con uno de dos tipo : Simple, o Pipelined.

10.6.2 - **Procesadores Array**

Replicar unidades funcionales es la tercera forma de incrementar performance. Existen muchas formas de replicar, y comenzamos por la más sencilla : los procesadores Array.

Un procesador Array típico consta de una unidad de instrucción que envía instrucciones a un conjunto de procesadores esclavos. Cada procesador esclavo ejecuta la instrucción que le fuera enviada interpretando las direcciones de los operandos como si fueran de su propia memoria.

Normalmente, se proveen instrucciones para permitir el intercambio de datos entre los procesadores en forma directa o indirecta.

Es común también que cada procesador tenga acceso a sus propios datos, permitiendo que diferentes direcciones sean accedidas por diferentes procesadores.

Un procesador array está directamente clasificado como una máquina abstracta con procesador de única instrucción, una sola memoria de instrucciones, múltiples procesadores de datos y múltiples memorias de datos.

Debido a que existen múltiples unidades funcionales debemos describir las formas en que están conectadas. Las conexiones entre las unidades funcionales se hacen mediante conmutadores (switches) abstractos, que pueden implementarse de diferentes forma : Buses, Switches dinámicos, Redes de interconexión estáticas.

Existen 4 formas de conectar unidades funcionales con switches abstractos, a saber :

*) **1-a-1**, una sola unidad funcional de un tipo está conectada a otra sola unidad funcional de otro tipo. Por supuesto que existen a nivel físico más de una conexión y la información fluye en ambas direcciones, como en la máquina Von Neumann descripta anteriormente. Esto no es realmente un switch, pero lo incluimos por completitud.

*) **n-a-n**, en esta configuración la i-ésima unidad de un conjunto de unidades funcionales está conectada con la i-ésima unidad de otro. Este tipo de switch es una conexión 1-a-1 que se repite n veces.

*) **1-a-n**, en esta configuración una unidad funcional está conectada a los n dispositivos de otro conjunto de unidades funcionales.

*) **n-por-n**, en esta configuración cada dispositivo de un conjunto de unidades funcionales puede comunicarse con cualquier dispositivo de un segundo conjunto de unidades y viceversa.

Todo procesador array tiene un switch de conexión de tipo 1-a-n que conecta el único procesador de instrucción a los procesadores de datos.

Se distinguen dos diferentes subfamilias, basándose en el tipo de switch utilizado para conectar los procesadores de datos a la jerarquía de memoria de datos.

La primera clase puede verse en la Fig. 10.5. Aquí la conexión del procesador-de-datos-memoria-de-datos

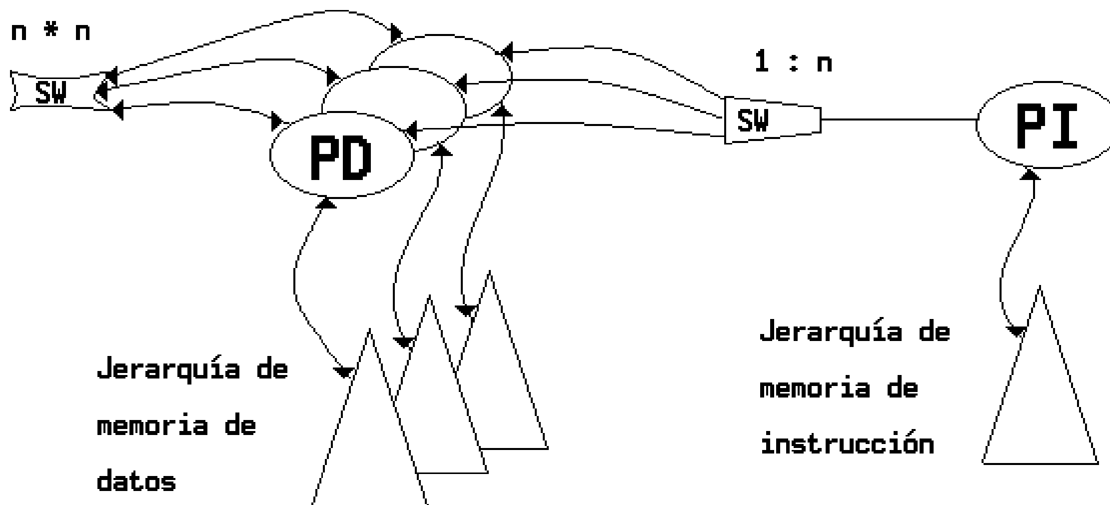


Fig. 10.5. - Un procesador array de Tipo 1.

es del tipo n-a-n y la conexión del procesador-de-datos-procesador-de-datos es del tipo n-por-n.

Este esquema de conexión se utilizó en máquinas tales como la Máquina de Conexión (The Connection Machine de Hillis, 1985).

El segundo tipo se ve en la Fig. 10.6, en este caso, el procesador-de-datos-memoria-de-datos están conectados con una conexión de tipo n-por-n y no existe conexión entre los procesadores de datos. Esta forma de switch es usualmente denominada una red de sincronización (alignement network) Un ejemplo de ésta lo constituye el Procesador Científico Burroughs (BSP).

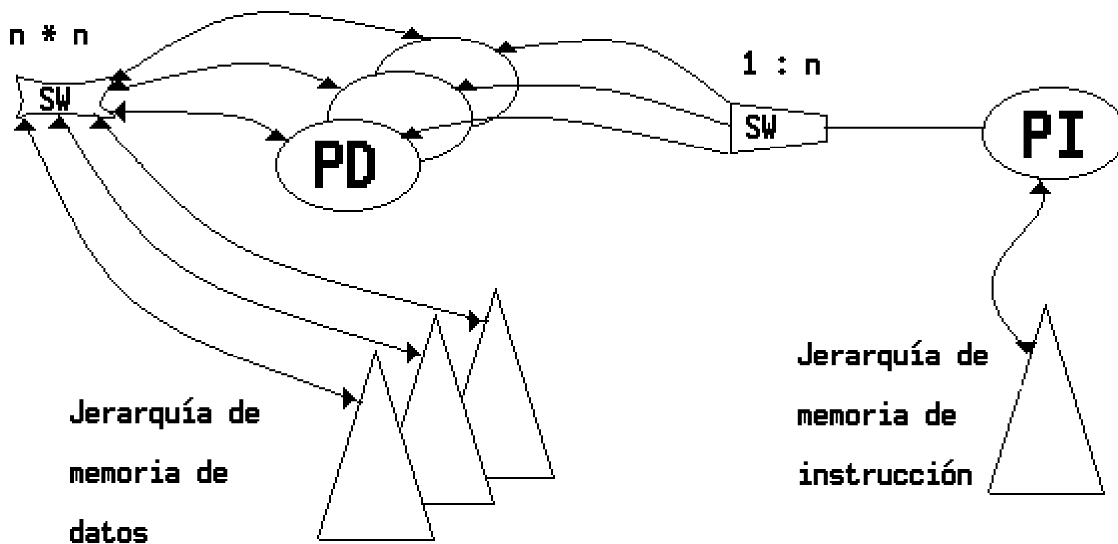


Fig. 10.6. - Un procesador array de Tipo 2.

Podemos introducir ahora nuestra taxonomía informalmente.

Clasificamos arquitecturas por el número de procesadores de instrucción que tienen, y por el número de procesadores de datos; por la cantidad de jerarquías de memorias que existen y por la forma en que estas unidades funcionales están conectadas por switches abstractos. Luego, un monoprocesador Von Neumann puede clasificarse como :

- La cantidad de procesadores de instrucción es uno.
- La cantidad de memorias de instrucción es uno.
- El switch entre el procesador de instrucciones y la memoria de instrucciones es de tipo 1-a-1.
- La cantidad de procesadores de datos es uno.
- La cantidad de memorias de datos es uno.
- El switch entre el procesador de datos y la memoria de datos es de tipo 1-a-1.
- El switch entre el procesador de datos y el procesador de instrucciones es del tipo 1-a-1.

Un procesador array de tipo 1 puede clasificarse como :

- La cantidad de procesadores de instrucción es uno.
- La cantidad de jerarquías de memoria de instrucción es uno.
- El switch entre el procesador de instrucciones y la memoria de instrucciones es de tipo 1-a-1.
- La cantidad de procesadores de datos es n.
- La cantidad de memorias de datos es n.
- El switch entre los procesadores de datos y las memorias de datos es de tipo n-a-n.
- El switch entre el procesador de instrucciones y los procesadores de datos es del tipo 1-a-n.
- El switch entre los procesadores de datos es del tipo n-por-n.

Veremos descripciones más complejas en la próxima sección. A medida que la cantidad de unidades funcionales crece, las posibilidades de interconexión crecen conjuntamente.

10.7. - **MAQUINAS PARALELAS VON NEUMANN**

Los procesadores arrays dependen de la existencia de que muchos datos puedan ser manejadas de la misma forma (igual instrucción) y al mismo tiempo. Muchos problemas no se encuadran adecuadamente en este paradigma.

Aún así, es natural considerar la replicación del procesador de instrucciones así como la del procesador de datos y crear múltiples líneas de control.

Esta es la idea que se plantea en las máquinas procesadores paralelas de Von Neumann.

De este supuesto surgen dos diferentes tipos de arquitectura : Sistemas fuertemente acoplados, y Sistemas débilmente acoplados.

Los sistemas fuertemente acoplados consisten en un conjunto de procesadores conectados a un conjunto de memorias mediante un switch dinámico. Cualquier procesador puede acceder cualquier posición de memoria con aproximadamente la misma latencia. La comunicación y sincronización entre los procesos se realiza mediante el uso de variables compartidas. Ejemplo de tales máquinas son la BBN Butterfly, la IBM 3081 y 3090 y la C.mmp.

Los sistemas débilmente acoplados consisten en un conjunto de procesadores, cada uno con su memoria local. La comunicación se produce por pedidos explícitos desde un procesador hacia otro a través de una red de interconexión. A través de intercambio de mensajes o por medio de llamadas a procedimientos remotos. Ejemplos de estas lo constituyen las máquinas hipercubo de Intel y el NCubo, los sistemas basados en el Transputer tales como el SuperNode (Proyecto Esprit 1985) y el Meiko MK40, así como viejos sistemas tales como el CM*.

En los sistemas fuertemente acoplados, tanto los procesadores de datos como los de instrucciones están duplicados, pero los procesadores de datos comparten una memoria común.

La correspondiente máquina abstracta puede verse en la Fig. 10.7. La mostramos aquí con múltiples memorias de datos y un switch de n-por-n entre los procesadores de datos y las memorias de datos debido a que es ligeramente más sugestiva que la implementación usual. Funcionalmente hablando ésta es indistinguible de una memoria común compartida.

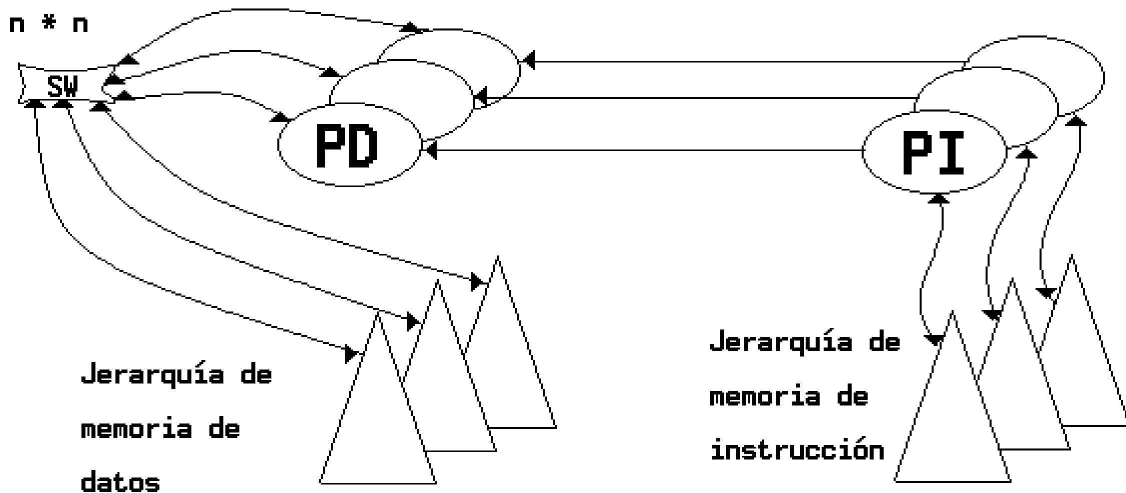


Fig. 10.7. - La máquina abstracta para un típico multiprocesador fuertemente acoplado.

Los sistemas débilmente acoplados también tienen duplicación de los procesadores de datos e instrucciones, pero los switches en el subsistema de datos difieren. Un débilmente acoplado típico puede verse en la Fig. 10.8. La conexión entre los procesadores de datos y las memorias de datos es de tipo n-a-n, y la conexión entre los procesadores de datos es del tipo n-por-n.

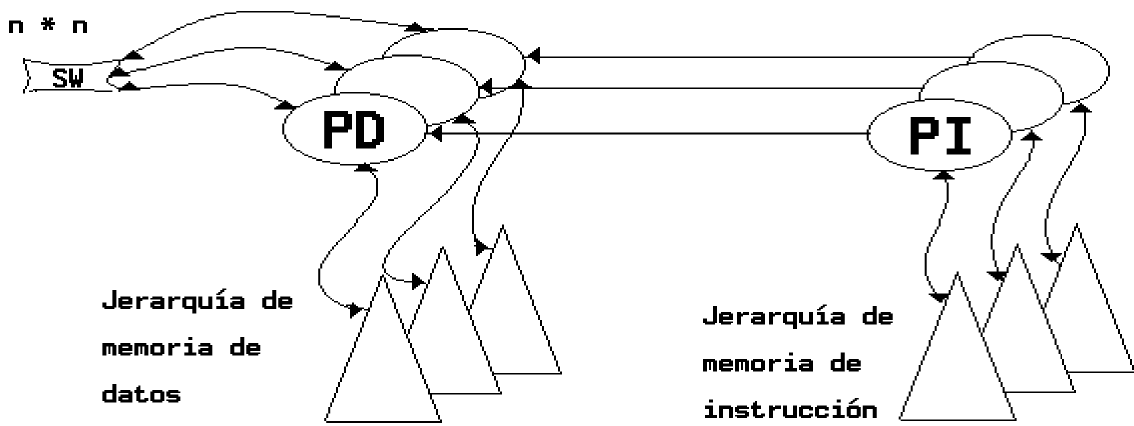


Fig. 10.8. - La máquina abstracta para un típico multiprocesador débilmente acoplado.

10.8. - <u>MAQUINAS QUE UTILIZAN OTROS MODELOS DE COMPUTO</u>

Las máquinas Von Neumann están todas basadas en el modelo de cómputo con líneas de instrucciones ejecutadas secuencialmente, excepto cuando el orden se altere explícitamente. Hablamos ya de que este ordenamiento es superrestrictivo. Esta dificultad empeora cuando se utilizan modelos de cómputo con múltiples líneas de control, ya que el programador debe considerar no solo el ordenamiento de las instrucciones en un conjunto en particular sino también las posibles secuencias de instrucciones en diferentes conjuntos interactuantes.

No sorprende entonces que aquellos involucrados en el diseño de las máquinas paralelas examinaran otros modelos de cómputo que no tuvieran esta tosca propiedad.

Estos nuevos modelos de cómputo están caracterizados por una completa ausencia de la descripción del programador en cuanto al orden de evaluación, que no sea el implícito por las dependencias de los datos. Esto permite que cualquier orden posible de evaluación para la ejecución pueda ser considerado, y el que otorgue la mejor performance pueda ser seleccionado por el compilador o el environment de runtime.

Los modelos de cómputo con esta propiedad se expresan usualmente en lenguajes funcionales.

10.8.1. - **Máquinas de reducción de grafos Graph Reduction Machines**

En reducción de grafos un cómputo se codifica como un grafo de funciones y argumentos que (recursivamente) tienen la propiedad de que el nodo raíz es un Aplicar, el subárbol izquierdo es la descripción de una función, y el subárbol derecho es la descripción de un argumento.

La descripción de una función o un argumento puede ser tanto un valor o una descripción de como obtener el valor.

Si ambos, la función y el argumento, han sido ya evaluados, entonces el subárbol (denominado Redex) se encuentra disponible para ejecución.

Cuando éste ha sido evaluado, sus valores reemplazan al subárbol. Si aún no ha sido evaluado, entonces algún subárbol debe ser Redex o el cómputo no puede proseguir.

Luego, en general las redex disponibles para ejecución se encuentran en las hojas del árbol. Típicamente existen múltiples redex disponibles para ser evaluadas en cualquier momento, por tanto pueden involucrarse múltiples procesadores en la evaluación simultáneamente.

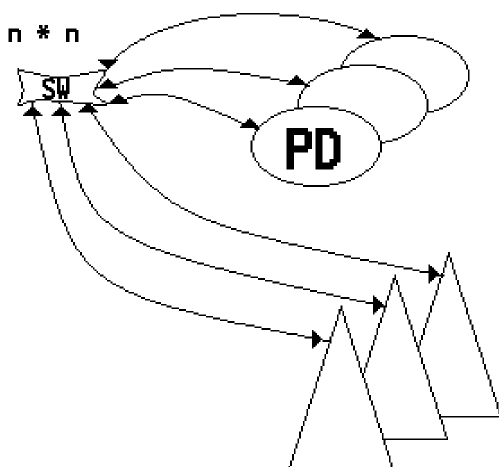


Fig. 10.9. - La máquina abstracta para reducción de grafos.

La máquina abstracta para reducción de grafos puede verse en la Fig. 10.9. Difiere primordialmente de la máquina abstracta de Von Neumann en la ausencia de una unidad de instrucciones y una memoria de instrucciones. El grafo que está siendo reducido juega aquí ambos roles, el de instrucciones (en su estructura) y el de datos (en sus contenidos).

El procesador de datos y sus memorias de datos asociadas son más parecidas a aquellos de un multiprocesador fuertemente acoplado, lo cual no debe sorprender ya que el grafo es una amplia estructura de datos compartidos.

Ya que no existe un procesador de instrucciones que provee las direcciones de los datos al procesador de datos, éste debe generarlas él solo.

Existen dos formas equivalentes de lograr esto.

El primero consiste en proveer al procesador de datos de un selector que seleccione cualquier redex disponible desde la memoria de datos.

El otro, es obtener una memoria de datos que actúe como un dispositivo activo que empuje los redex disponibles hacia el procesador de datos.

Los trabajos en reducción de grafos han tenido extremada importancia en Europa y el Reino Unido y han sido bastante exitosos. Varias máquinas han sido diseñadas y construidas, algunos ejemplos son Alice (1981) y Flagship (1988).

10.8.2. - **Máquinas Dataflow**

Las máquinas dataflow son otra clase de máquinas que no cuentan con un mecanismo secuenciador de instrucciones sino que está implícito por las dependencias de los datos.

Su modelo de cómputo es un grafo con arcos dirigidos, a lo largo de los cuales fluyen los datos y los vértices representan las operaciones que transforman los datos.

Ciertos arcos tienen vértice sólo de un lado; estos representan los inputs o outputs del cómputo.

Un vértice, u operador, se dispara de acuerdo a alguna regla de encendido (firing rule) especificada por la forma particular de dataflow.

En general, la detonación consume un grupo de datos de cada arco de input y produce un resultado que parte luego a través de los arcos de salida.

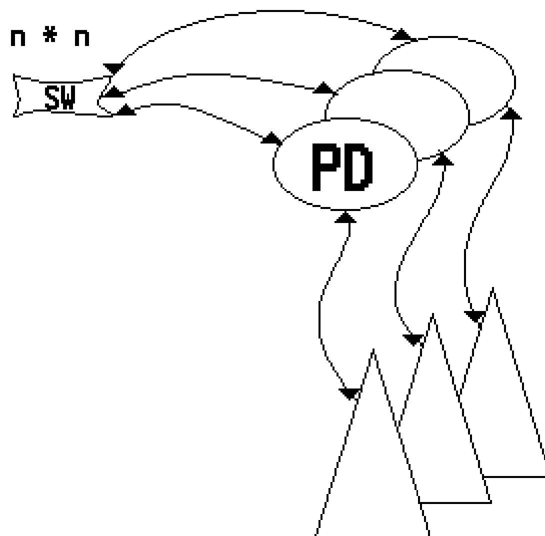


Fig. 10.10. - La máquina abstracta para un Dataflow (de tipo Dennis).

Muchas de las máquinas de dataflow están basadas en un anillo que consiste de un almacenamiento comparativo (donde los valores de los datos esperan hasta que el conjunto de todos los operandos estén presentes), una memoria que contiene los operadores y una serie de elementos de procesamiento que ejecutan los operadores, produciendo los valores del resultado que fluyen alrededor del anillo hacia el almacenamiento comparativo.

La máquina abstracta para un procesador dataflow puede tener dos formas, que se corresponden a las dos formas en que pueden disponerse los procesadores de datos en un procesador array.

En la primera forma, todas las memorias de datos son visibles a todos los procesadores. Esta visión es la que se aplicó en la máquina Dataflow Manchester (1980) y en la máquina de Arvind.

Esta forma de procesamiento es funcionalmente idéntica a las máquinas paralelas de reducción de grafos y, por tanto, tienen el mismo diagrama de máquina abstracta (Fig. 10.9).

En la segunda forma, que puede verse en la Fig.10.10, cada procesador tiene su propia memoria local, y los datos requeridos por otro procesador fluyen a través de switches entre-procesadores-de-datos.

La segunda forma es quizás más intuitiva; en el límite cada procesador ejecuta sólo un único operador.

10.9. **EL MODELO FORMAL**

Nuestro esquema de clasificación consiste de dos niveles de detalle y discriminación. El primer nivel define una arquitectura especificando :

- cantidad de procesadores de instrucción (PI)
- cantidad de memorias de instrucción (MI)
- el tipo de switch de conexión de los PIs a las MIs
- la cantidad de procesadores de datos (PD)
- la cantidad de memorias de datos (MD)
- el tipo de switch de conexión de los PDs y las MDs.
- el tipo de switch de conexión entre los PIs y los PDs
- el tipo de switch de conexión entre los PDs.

El primer nivel hace un refinamiento de la clasificación de Flynn expandiendo las clases de SIMD y MIMD a un subconjunto de subclases que capturan importantes arquitecturas existentes.

El segundo nivel permite refinamientos ulteriores describiendo si se puede o no pipelinizar el procesador y hasta qué punto, y dando el comportamiento del diagrama de estado de los procesadores. Es posible un tercer nivel que describa detalles de implementación. Esta clasificación se muestra en la Fig. 10.11.

Como hemos visto esta clasificación captura las diferencias entre arquitecturas que son significativamente diferentes, ignorando diferencias que son una cuestión primordial en la implementación.

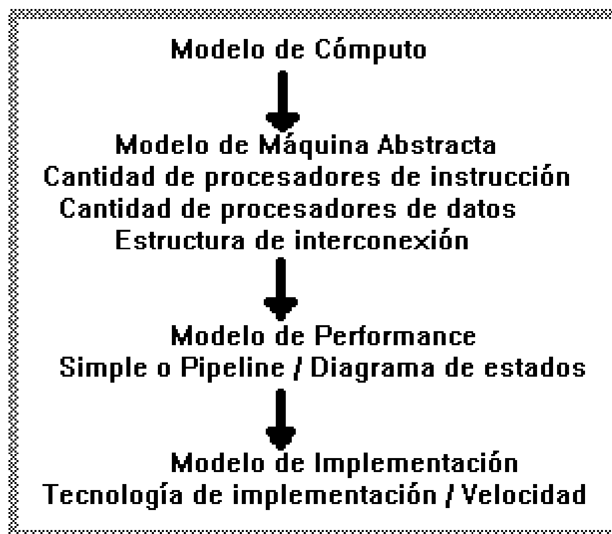


Fig. 10.11. - El método de clasificación

La Tabla de la Fig. 10.12 muestra un conjunto sencillo de arquitecturas posibles basándonos en el supuesto de que la cantidad de memorias coincide con la cantidad de procesadores para cada subsistemas.

A pesar de que existen arquitecturas interesantes en las cuales este supuesto no se cumple, por ahora, en aras de la claridad, las ignoraremos.

Las clases 1 a 5 son las máquinas dataflow/reducción que no tienen instrucciones en el sentido usual de la palabra. La clase 6 es el monoprocesador de Von Neumann. Las clases 7 a 10 son procesadores array.

Nótese que las clases 5 a 10 representan extensiones de las arquitecturas a las que pertenecen de forma tal que cuentan con una doble conectividad para sus procesadores de datos.

Las clases 11 y 12 son las arquitecturas MISD. A pesar de que estas clases han sido tratadas como una aberración, existen lenguajes para los cuales este tipo de ejecución es apropiada. Por ejemplo, en NIAL (Nested Interactive Array Language, 1986) se puede escribir :

[f g h] x

que es la aplicación paralela de las funciones f, g, y h sobre x.

No estamos enterados de ninguna arquitectura existente de este tipo, sin embargo el concepto no es totalmente ridículo.

Las clases 13 a 28 son los multiprocesadores de varios tipos. Las clases 13 a 20 son más o menos los multiprocesadores convencionales con diferentes formas de la estructura de conexión.

Las clases 21 a 28 son arquitecturas más exóticas en las cuales las conexiones entre los procesadores de instrucciones es del tipo n-por-n. Estas clases se encuentran completamente inexploradas.

Las arquitecturas que tienen una conexión n-por-n entre los procesadores de instrucciones y las memorias de instrucciones son aquellas en donde la decisión del procesador de ejecutar una instrucción particular se realiza tarde (justo antes de la ejecución).

Las arquitecturas Dataflow y de Reducción de grafos tienen a menudo esta propiedad, pero la única máquina Von Neumann de la que tengamos conocimiento que se comporta de esta forma es el Procesador de Elementos Heterogéneos (1981) de la compañía Denelcor.

Algunas máquinas Von Neumann que existen cuentan con más de un sólo procesador de datos. Por ejemplo, algunos sistemas pipeline de alta performance (Cray I, Cyber 205) tienen un procesador de datos para instrucciones vectoriales y otro para instrucciones escalares. Algunos tienen inclusive un procesador escalar separado para las instrucciones de este tipo.

Las arquitecturas de este tipo no concuerdan con el esquema sencillo presentado antes, pero pueden representarse fácilmente incluyendo las unidades de procesamiento vectorial conjuntamente con los procesadores de datos (la cantidad de procesadores de datos es 1 + m).

Fig. 10.12. - TABLA DE ARQUITECTURAS POSIBLES

Clase	PIs	PDs	PI-PD	PI-MI	PD-MD	PD-PD	Nombre
1	0	1	ninguna	Ninguna	1-1	ninguna	Monoprocesador reducción/dataflow
2	0	n	ninguna	Ninguna	n-n	ninguna	Máquinas separadas
3	0	n	ninguna	Ninguna	n-n	n*n	Débilmente acoplado reducción/dataflow
4	0	n	ninguna	Ninguna	n-n	ninguna	Fuertemente acoplado reducción/dataflow
5	0	n	ninguna	Ninguna	n*n	n*n	
6	1	1	1-1	1-1	1-1	ninguna	Monoprocesador Von Neumann
7	1	n	1-n	1-1	n-n	ninguna	
8	1	n	1-n	1-1	n-n	n*n	Array processor Tipo 1
9	1	n	1-n	1-1	n*n	ninguna	Array processor Tipo 2
10	1	n	1-n	1-1	n*n	n*n	
11	n	1	1-n	n-n	1-1	ninguna	
12	n	1	1-n	n*n	1-1	ninguna	
13	n	n	n-n	n-n	n-n	ninguna	Monoprocesadores separados Von Neumann
14	n	n	n-n	n-n	n-n	n*n	Von Neumann débilmente acoplados
15	n	n	n-n	n-n	n*n	ninguna	Von Neumann fuertemente acoplados
16	n	n	n-n	n-n	n*n	n*n	
17	n	n	n-n	n*n	n-n	ninguna	
18	n	n	n-n	n*n	n-n	n*n	
19	n	n	n-n	n*n	n*n	ninguna	Procesador de elementos Heterogéneos de Denelcor
20	n	n	n-n	n*n	n*n	n*n	
21	n	n	n*n	n-n	n-n	ninguna	
22	n	n	n*n	n-n	n-n	n*n	
23	n	n	n*n	n-n	n*n	ninguna	
24	n	n	n*n	n-n	n*n	n*n	
25	n	n	n*n	n*n	n-n	ninguna	
26	n	n	n*n	n*n	n-n	n*n	
27	n	n	n*n	n*n	n*n	ninguna	
28	n	N	n*n	n*n	n*n	n*n	

10.10. - **OTRA PROPUESTA TAXONOMICA**.

La clasificación anterior adolece de ciertos defectos que fueron observados por Subrata Dasgupta (1990) quien propuso a su vez un esquema taxonómico más completo utilizando parte de las características definidas por Skillicorn.

10.10.1. - **TAXONOMIA JERARQUICA**.

Una clasificación de tipo jerárquica provee las bases para comparar y discriminar distintos objetos. Esta clasificación permite además determinar puntos de convergencia a través de los niveles de categorías, según se puede apreciar en el diagrama de la Fig. 10.13.

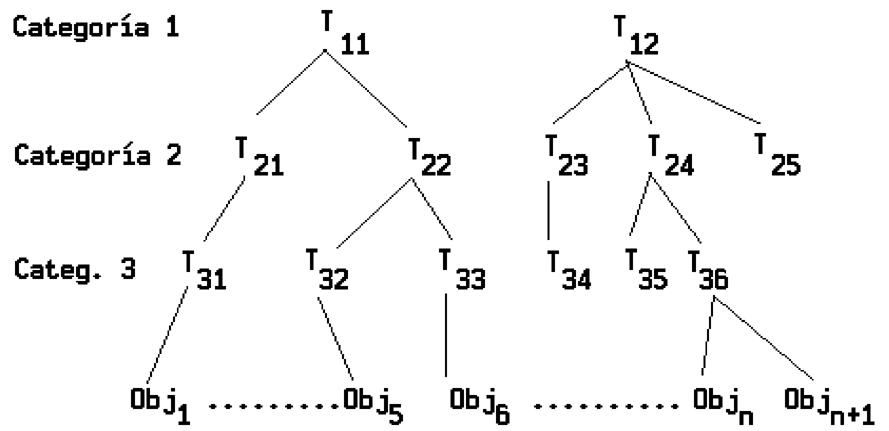


Fig. 10.13.

10.10.2. - **Limitaciones de la taxonomía de Skillicorn**.

Vamos a denominar **TC** (Caracteres Taxonómicos) a los PI, PD, etc.

Las limitaciones de la taxonomía de Skillicorn están dadas por :

Predecir Capacidad (Poder) : De acuerdo a la clasificación de Skillicorn, no es posible determinar, de antemano, cuando una arquitectura es más poderosa que otra, o similar, sin comparar los valores de sus respectivos TC. Esto se debe a que esta clasificación no es jerárquica y entonces tiene una sola categoría.

Explicar Capacidad (Poder): Si bien, dando las características precisas de los PI y PD, existe la posibilidad de explicar las capacidades de las arquitecturas, esto no se realiza completamente.

La clasificación no incluye el concepto de Pipelining en el mismo nivel de abstracción que lo hace con las memorias, procesadores y switches. Sólo aparece en el diagrama de estados. Pero, los diagramas de estados no identifican formalmente taxonomías y no constituyen categorías.

Además, como la clasificación no es de tipo jerárquica, existen más elementos en el nivel superior que en el inferior, ver Fig. 10.14.

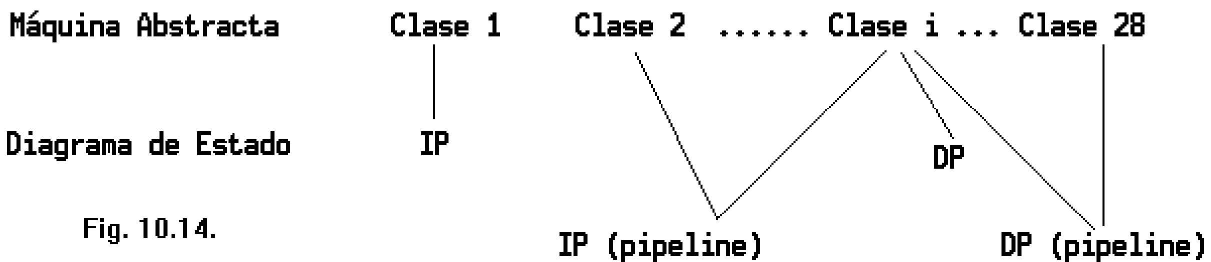


Fig. 10.14.

En cuanto a las memorias (MI, MD), se dice que son jerárquicas, pero no se discrimina entre memoria principal, cache, etc. y no se tiene en cuenta su importancia en el multiprocesamiento.

10.10.3. - **Nueva Taxonomía**.

Se toma como punto de partida el esquema de Skillicorn, pero teniendo en cuenta sus limitaciones se llegará a un resultado distinto.

La nueva taxonomía se construye con siete (7) **primitivas** de endoarquitectura que llamaremos **átomos**.

Atomos del mismo tipo pueden ser combinados en entidades más complejas que llamaremos **radicales atómicos**, que a su vez pueden ser combinados en conceptos más complicados llamados **radicales no-atómicos**.

Los radicales no-atómicos pueden ser combinados en **moléculas**, las que denotarán entidades de endoarquitectura completas.

10.10.3.1. - **Atomos**.

Usaremos los siguientes símbolos:

- iM : memoria interleaving
- sM : memoria simple
- C : cache
- sI : unidad de instrucciones simple
- pI : unidad de instrucciones pipeline
- sX : unidad de ejecución simple
- pX : unidad de ejecución pipeline

Las letras (M, **I**, C, X) son **iones**.

Las letras minúsculas (s, i, p) son **prefijos** de los iones.

Las funciones de los iones son las conocidas, pero aquí entenderemos que las del ión **I** son :

- Determinar próxima instrucción.
- Cargar instrucción de M o C.
- Decodificar instrucción.
- Calcular dirección de operandos.
- Transferir direcciones de operandos a X.
- Recibir estados de X.

Y las funciones del ión X son :

- Recibir direcciones de operandos y operadores (instrucciones) de I.
- Cargar operandos desde M o C.
- Ejecutar instrucciones.
- Almacenar resultados en M o C.
- Informar estado a I.

De lo que resulta que las funciones de I y X son equivalentes a las de los PI y PD de Skillicorn.

10.10.3.2. - **Radicales Atómicos**.

Son las instancias replicadas de un átomo, lo que se escribirá como A_n donde el subíndice n, que llamaremos el número de radical (**RN**), puede ser una constante o variable pero siempre dentro del rango de los enteros positivos.

Por ejemplo:

*) iM_2 denota dos átomos de memoria interleaving
*) sM_4 denota 4 átomos de memoria
*) pI_m denota m instancias de procesadores. de instrucciones con pipeline

Si RN = 1 se dice que es un radical monoatómico y es equivalente escribir C o C_1.
si RM >> 1 se dice que es un radical multiatómico.

10.10.3.3. - **Radicales No-atómicos**.

Es la combinación de átomos distintos. Por convención se escribirá siempre más a la izquierda los correspondientes a memoria (M), luego los de cache (C), y por último los correspondientes a procesador (P).

Por ejemplo CP indica un procesador con cache, que es una combinación de un radical C con un I, o un X.

También aquí puede haber replicaciones, y obtenerse ejemplos como este :

$iM.(C.sI_2)_k$

que indica una memoria de instrucciones combinada con k instancias de dos procesadores de instrucciones con cache. En este caso el radical es de tipo (MCP-).

10.10.3.4. - **Moléculas**.

Una molécula I- es un radical MCP- que representa un subsistema completo de "preparación de instrucciones" dentro del nivel de endoarquitectura.

Una molécula X- es un radical MCP que representa un subsistema completo de "ejecución de instrucciones" dentro del nivel de endoarquitectura.

Una macromolécula es una combinación de moléculas I- o X- que representan una computadora completa a nivel de endoarquitectura. Por convención se escribirá I a la izquierda de X.

10.10.3.5. - **Sintaxis**.

M' ::= $iM \mid sM \mid iM_n \mid sM_n$
C' ::= $C \mid C_n$
I' ::= $sI \mid pI \mid sI_n \mid pI_n$
X' ::= $sX \mid pX \mid sX_n \mid pX_n$
CP' ::= C'.I' | C'.X' | C'.CP' | $(CP)'_n$
MCP'::=M'.I' | M'.X' | M'.MCP' | $(MCP)'_n$
I'' ::= MCP'
X'' ::= MCP'
MM'' ::= (I'') (X'') | $((I'')(X''))_n$

Atomos : iM, sM, C, sI, pI, sX, pX.
Instancias : subíndice n
$(.)_n$: instancias de radicales o moléculas (replicación)
.' : radical
.'' : moléculas

10.10.4. - **Estructuras de Radicales y Moléculas**.

Si decimos que R es un radical, podemos escribir cabeza(R) y cola(R). Por ejemplo si R1 = C.sI, entonces cabeza (R1) = C y cola (R1) = sI.

Si R2 = $(C.pI)_n$, resulta cabeza(R2) = cola(R2) = $(C.pI)_n$.

Si R3 = $iM_m.(C.pI)_n$, resulta cabeza(R3) = iM_m y cola(R3) = $(C.pI)_n$.

La estructura de un radical (y obviamente de una molécula I o X) puede ser determinada por dos operadores **Rep** y **Link** que se definen así:

Rep(R) causa

R
.
.
R

Link(R1,R2) causa las siguientes estructuras:

- si cola(R1) y cabeza(R2) tienen una sola instancia, entonces

R ——— R   o sea un camino dedicado

- si cola(R1) tiene una sola instancia pero cabeza(R2) tiene varias (en particular tomemos que sean 2), entonces

R1 ——< Z, Z   caminos compartidos

- si cola(R1) tiene varias instancias (w) y cabeza(R2) solo una, entonces

W, W >—— R2   caminos alternativos

- si cola(R1) tiene varias instancias (w) y cabeza(R2) también (z), entonces

W, W >——< Z, Z   caminos compartidos entre todas las instancias

Usando Rep y Link, la estructura de cualquier macromolécula puede ser determinada de acuerdo a las reglas 1) a 9) que se muestran a continuación.

Nota: St(x) significa estructura de x y $St^{-1}(x)$ es la imagen espejada de la estructura de x.

1) St(M')	=	if iM	then iM
		if sM	then sM
		if iM_n	then Rep(iM)
		if sM_n	then Rep(sM)
2) St(C')	=	if C	then C
		if C_n	then Rep(C)
3) St(I')	=	if sI	then sI
		if pI	then pI
		if sI_n	then Rep(sI)
		if pI_n	then Rep(pI)
4) St(X')	=	if sX	then sX
		if pX	then pX
		if sX_n	then Rep(sX)
		if pX_n	then Rep(pX)
5) St(CP')	=	if C'.I'	then Link(St(C'),St(I'))
		if C'.X'	then Link(St(C'),St(X'))
		if C'.CP'	then Link(St(C'),St(CP'))
		if $(CP')_n$	then Rep(St(CP'))
6) St(MCP')	=	if M'.I'	then Link(St(M'),St(I'))
		if M'.X'	then Link(St(M'),St(X'))
		if M'.CP'	then Link(St(M'),St(CP'))

		if M'.MCP'	then Link(St(M'),St(MCP'))
		if $(MCP')_n$	then Rep(St(MCP'))
7) St(I'')	=	St(MCP')	
8) St(X'')	=	St(MCP')	
9) St(MM'')	=	if (I'')(X'')	then St(I'') | St^{-1}(X'')
		if $(I'')(X'')_n$	then Rep(St(I'') | St^{-1}(X''))

Para comprender mejor lo anterior y ver que los switches de Skillicorn están implícitos en las fórmulas veamos algunos ejemplos:

Fórmula	**Estructura**
C.sI	C — sI
[C.pX]n	C — pX ... . C — pX
Cm.sXn	C sX .. .. C sX
C.[C2.pI]2	C ... C — pI C C ... C — pI
iMm.[C.pI]n	iM C — pI ... ... iM C — pI
[sM.pI][sM.C.pX4]	sM-pI | pX ... ... pX — C — sM

10.10.5. - **Categorías Jerárquicas del sistema taxonómico**.

Según este esquema tendremos 3 principales jerarquías.

1)- La categoría MCP, es la molecular y la más baja.

2)- La categoría CP, la siguiente.

3)- La categoría P, es la más alta, correspondiendo a los procesadores.

Veamos como ejemplo (Fig. 10.15) el caso de la VAX 8600, Vax 11/780 y VAX 11/750.

Para asignar un lugar a una computadora en esta clasificación, es necesario establecer primero su fórmula molecular correspondiente a su endoarquitectura.

Esta se obtiene en términos de memoria, cache, procesadores (X e I), si hay interleaving de memoria o no, si los procesadores son de tipo pipeline y si la cache retiene datos, instrucciones o ambos.

Es necesario conocer la cantidad de estos elementos y sus interconexiones.

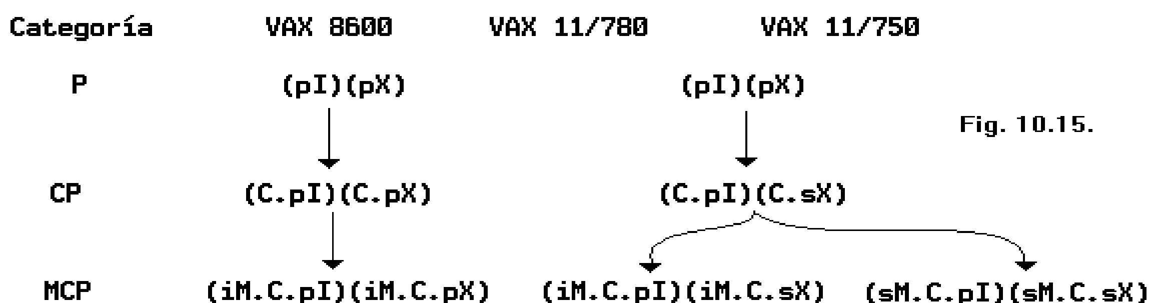


Fig. 10.15.

Toda esta información puede ser obtenida en forma directa de la misma fórmula. Tomemos como ejemplo el caso de la VAX 8600, cuya fórmula es:

(iM.C.pI)(iM.C.pX)

De aquí se infiere que es monoprocesador, que ejecuta en pipeline el ciclo completo de sus instrucciones, que el flujo del pipe se ve ayudado por memoria cache y memoria interleaving (repetidas ambas).

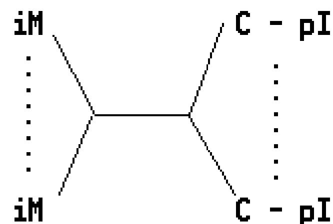
La categoría CP es una abstracción de la MCP (molecular) e identifica a las computadoras sin tener en cuenta los componentes de memoria.

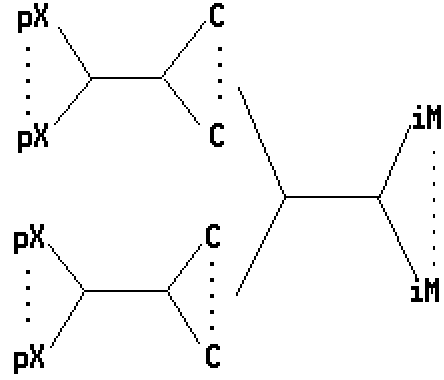
La categoría P es una abstracción de la CP y clasifica procesadores en función de cómo ejecutan instrucciones, independiente de la presencia de caches.

10.10.5.1. - **Algunos ejemplos**.

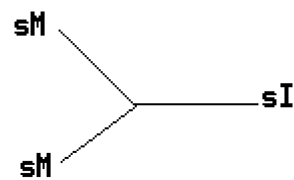
Illiac IV	$(sM_{64}.sI)(sM.sX)_{64}$
Cray X-MP	$(iM_m.(C.pI)_n)(iM_m.(C_r.pX_s)_q)$
IBM 3838	$(sM.pI)(sM.pX_7)$
Manchester	$(sM.sI)(sM.sX_n)$

Cray X-MP





Illiac IV



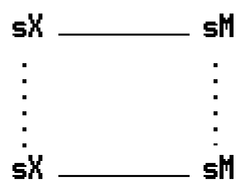


Fig. 10.16. - Gráficos de algunos ejemplos.

En el caso de la Manchester es discutible la presencia de sI, pero si se elimina, no queda clara la búsqueda de instrucciones, o sea que en esta clasificación las máquinas DF (Dataflow) tampoco quedan bien caracterizadas.

EJERCICIOS

1) Cuál es el problema específico de la clasificación de Flynn que llevar a plantear un nuevo método de clasificación de las arquitecturas de computadores.
2) Qué es la máquina abstracta y cuál es su relación con el modelo de cómputo ?
3) Cuáles son los tipos de unidades funcionales que se emplean en esta clasificación y cuál es su función ?
4) Cuáles son las funciones del Procesador de Instrucción en la máquina abstracta Von Neumann ?
5) Qué es un diagrama de estados y cuando se hace necesario su uso ?
6) Cuáles son las funciones del Procesador de Datos en la máquina abstracta de Von Neumann ?
7) Qué significa que la comunicación entre el PI y el PD debe ser sincrónica ? Justifique.
8) Cómo se conceptualiza en esta clasificación la característica de las jerarquías de memorias ?
9) Cuáles son las formas de incrementar performance ? Explique cada una.
10) Cuáles son las formas de conectar unidades funcionales ?
11) Explique las diferencias entre las dos clases posibles de procesadores array.
12) Cuál es la diferencia conceptual de la máquina abstracta de un sistema computador fuertemente acoplado con la de un sistema débilmente acoplado ?
13) Porqué una arquitectura de reducción de grafos carece de PI y de MI ? Justifique.

14) Cómo se proveen las direcciones de los datos en un sistema de reducción de grafos ?
15) Relacione el "anillo comparativo" de la máquina dataflow en este capítulo con lo visto en el capítulo 9.
16) Porqué se pueden tener dos modelos de máquinas abstractas para la arquitectura Dataflow ? Justifique.
17) Formalmente, qué se especifica en el 1er nivel de esta clasificación ?
18) Idem 17) para el 2do y 3er nivel.
19) Cómo se pueden encuadrar en esta clasificación las arquitecturas con procesadores vectoriales ? Justifique.

CAPITULO 11

SISTEMAS OPERATIVOS - INTRODUCCION

Un Sistema Operativo es un programa que actúa como interfase entre el usuario de una computadora y el hardware de la misma. El propósito es proveer un entorno en el cual el usuario puede ejecutar programas.

O sea que un objetivo principal de un SO es hacer que un sistema sea CONVENIENTE de usar. Otro objetivo es que se use el hardware de manera EFICIENTE, o sea que está íntimamente ligado a él. Un SO es un programa para un determinado hardware.

Otro objetivo es administrar los recursos de una computadora, tanto de hardware como de software. Por lo tanto, un SO es un conjunto de programas que administran un sistema.

Para entender qué son los SO hay que entender como fueron desarrollándose, lo cual se explica en este apunte.

11.1. ¿ QUE ES UN SO ?

Como sabemos, el hardware provee los recursos básicos de un sistema de computación. Mediante los programas de aplicación se definen las formas en que se usan esos recursos para resolver problemas para los usuarios. Hay diversos programas de aplicación. Un sistema operativo controla y coordina el uso del hardware entre los distintos programas de aplicación para los diversos usuarios.

El Sistema Operativo provee los medios para utilizar de forma correcta los recursos de un sistema de computación; no hace ninguna función útil por sí mismo. Simplemente provee un entorno en el cual otros programas pueden hacer trabajo útil.

Podemos ver al SO como un ASIGNADOR DE RECURSOS. Un sistema tiene diversos recursos que pueden requerirse para resolver un problema: tiempo de CPU, espacio de memoria, espacio de almacenamiento de archivos, procesos, dispositivos de E/S, etc. El SO actúa como el gerente de esos recursos y se los otorga a programas específicos y usuarios a medida que son necesarios. Como puede haber diversos pedidos conflictivos para recursos, el SO tiene qué decidir que pedidos son atendidos y cuáles no. Su fuente principal de trabajo es un conjunto de tablas en las cuales se describen los recursos y a que procesos están asignados los mismos.

Otro punto de vista de los SO apunta a la necesidad de controlar los distintos dispositivos de E/S y programas del usuario. Un SO es un programa de control. Un programa de control controla la ejecución de programas de usuario para prevenir errores y uso impropio de la computadora. Está relacionado especialmente con la operación y control de dispositivos de E/S.

Para ver qué son los SO, estudiemos como fueron construidos en los últimos 30 años. Viendo esta evolución, podemos identificar elementos en común, y cómo fueron mejorando.

Los SO y la arquitectura de los computadores tienen una gran influencia el uno sobre el otro. Para facilitar el uso del hardware se construyeron los SO. A medida que se diseñaron y usaron SO, se volvieron obvios ciertos cambios en el diseño del hardware que simplificaron los sistemas operativos. En esta revisión histórica veremos como la introducción de nuevo hardware es la solución natural para muchos de los problemas de sistemas operativos.

11.2. LOS PRIMEROS SISTEMAS

Inicialmente solo hubo hardware. Las primeras computadoras eran máquinas muy grandes que se programaban desde una consola. El programador podía escribir un programa, y luego operar el programa directamente desde la consola del operador.

Primero, el programa tenía que cargarse manualmente en la memoria, ya sea por medio de switches (llaves de conmutación), cinta de papel, o tarjetas perforadas. Luego, se apretaban los botones apropiados para cargar la dirección de comienzo y comenzaba la ejecución del programa. A medida que el programa corría, el programador/operador podía monitorear su ejecución por medio de luces en la consola.

Si se descubrían errores, el programador podía parar el programa, examinar los contenidos de la memoria y registros, y corregir el programa directamente desde la consola. La salida se imprimía o perforaba en cintas o tarjetas para una impresión posterior.

Un aspecto importante de este entorno era la naturaleza interactiva HANDS-ON. El programador era el operador. Muchos sistemas usaban un esquema de reserva para otorgar tiempo de maquina: para usar la maquina se pedía un turno en una hoja de papel.

Esta aproximación tiene ciertos problemas: supongamos que uno reserva una hora, y necesita más que ese tiempo para hallar un error. Uno tiene que parar, recolectar lo que se pueda, y volver mas tarde para continuar (consideremos que no existían ensambladores, y mucho menos compiladores). Por otro lado, uno podía terminar antes de que se terminara su tiempo, quedando el resto del tiempo la máquina sin ser usada.

A medida que pasó el tiempo, se construyó software y hardware adicional. Lectoras de tarjeta, impresoras de línea y cintas magnéticas se convirtieron en lo más común. Se diseñaron assemblers, cargadores y linkers para

facilitar la programación, así como también bibliotecas de funciones comunes. Estas podían copiarse en un nuevo programa sin tener que escribirse dos veces.

Las rutinas que hacen entrada y salida son especialmente importantes. Cada dispositivo de E/S nuevo tiene sus características propias, requiriendo una programación cuidadosa. Para cada dispositivo se escribe una subrutina especial, llamada el manejador del dispositivo (device driver). Este programa conoce como deben usarse los buffers, banderas, registros, bits de control y estado para un dispositivo en particular. Cada tipo distinto de dispositivo tiene su propio manejador. En vez de escribir el código necesario cada vez, el device driver se utilizaba directamente de la biblioteca.

Posteriormente aparecen compiladores (Fortran, Cobol, etc.) que facilitan la tarea de programación, pero dificultan la tarea de operación de la computadora. Para preparar un programa Fortran para ejecución había que ejecutar los siguientes pasos:

- Montar la cinta magnética o cargar las tarjetas perforadas que contenían el Compilador Fortran.
- Cargar el compilador Fortran.
- Leer el código fuente del programa de tarjetas perforadas y escribirlo en otra cinta magnética.
- Compilar el programa. El compilador Fortran producía una salida en lenguaje ensamblador.
- Montar la cinta que contiene el Assembler.
- Ensamblar el programa.
- Linkeditar la salida del programa para incluir rutinas de bibliotecas y subrutinas.
- Cargar el programa ejecutable y ejecutarlo, para su corrección.

Para ejecutar un trabajo hay una gran cantidad de tiempo de preparación (setup time), debido a la gran cantidad de pasos de que consta el trabajo. Si ocurre un error en algún paso, hay que comenzar nuevamente desde el comienzo, teniendo que cargar cintas o tarjetas.

11.3. MONITOR SIMPLE O SISTEMA BATCH SENCILLO

Vimos que el tiempo de preparación es un problema importante. Durante el tiempo de montado de cinta, o mientras el operador trabaja en la consola, la CPU esta desocupada. En esos tiempos, un computador era muy caro, con un tiempo de vida esperado corto. Por lo tanto, el tiempo de computación era muy valioso, y los propietarios querían utilizarlo lo más posible.

Hubo una solución doble: la primera fue contratar operadores profesionales, de tal forma que el programador no operara más la computadora, y eliminar el tiempo desperdiciado por las reservas. Los operadores no saben corregir los programas: lo único que hacen es obtener un vuelco de memoria y registros para el programador, y así poder seguir trabajando.

La segunda solución trató de reducir el tiempo de preparación. Los trabajos con necesidades similares eran loteados (batched) juntos, como un solo grupo. Si el operador recibe varios trabajos Fortran y otros tantos Cobol, en vez de ejecutarlos en orden de llegada ejecuta los programas Fortran juntos, y los Cobol juntos, y se gana el tiempo de carga de los compiladores.

A pesar de mejorar la utilización, sigue habiendo problemas: si para un programa, el operador tiene que mirar la consola para ver la condición de parada, si fue un error, hacer un dump (vuelco de memoria), y luego cargar la lectora con el próximo trabajo. Durante este tiempo, la CPU se mantiene sin uso.

Para solucionar este problema se introduce el secuenciamiento automático de trabajos, y con él, se crean los primeros sistemas operativos rudimentarios. La idea es que un programa pequeño, llamado monitor residente, transfiera automáticamente el control de un trabajo al siguiente.

Inicialmente, el monitor tiene el control del computador. El mismo transfiere el control a un programa. Cuando éste termina, retorna el control al monitor, que dará control al próximo programa. El monitor tiene que saber qué programa ejecutar. Con este fin se introducen las tarjetas de control, para dar información directamente al monitor. Estas son tarjetas especiales que se mezclan con las de datos o programa, y son directivas que indican qué programa se ejecutará y qué recursos necesitará.

Las tarjetas de control tienen una identificación especial para diferenciarlas de las de datos (//, $, *, ?). P. ej.:

* TRABAJO
* EJECUTAR
* DISCO
* FIN

Los errores son atrapados (interrupciones), y reciben un tratamiento adecuado (por ejemplo, un dump), y luego se sigue ejecutando la próxima tarjeta de control.

Para evitar que un programa, por error, lea tarjetas de control del siguiente trabajo, se utiliza un manejador de dispositivos (o sea que el encargado de leer es el

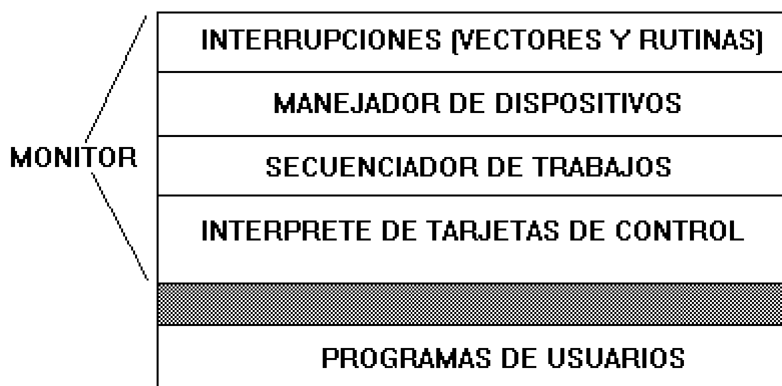


Fig. 11.1. - Esquema de Monitor.

Monitor), y esta instrucción (lectura) es una instrucción privilegiada.

Para hacer un pedido al SO, existen las llamadas al supervisor (SVC en IBM, Trap en PDP, nuestras BSV -bifurcación al supervisor-). Estas provocan una interrupción. En la Fig. 11.2 vemos un ejemplo simple de una lectura.

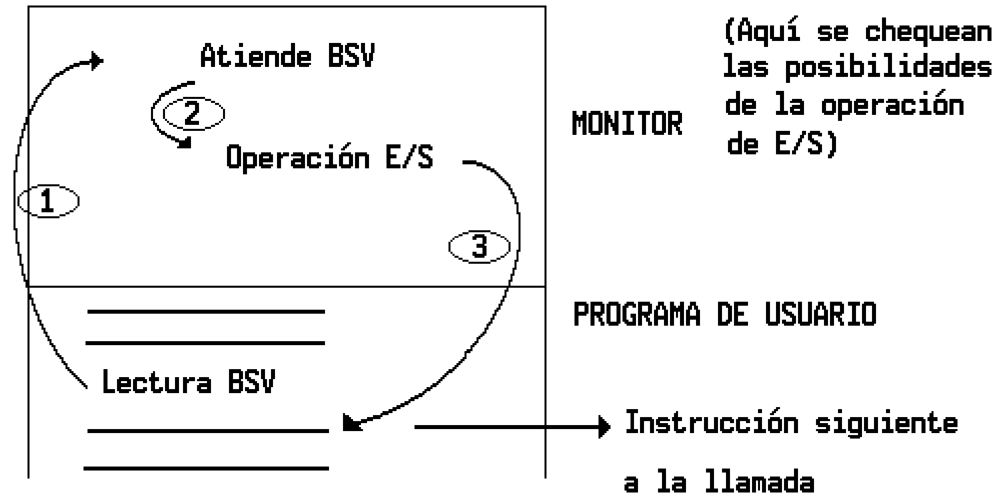


Fig. 11.2. - Operación modo Maestro/Esclavo.

Podemos ver que un monitor residente tiene varias partes identificables.

Una principal es el intérprete de tarjetas de control. Este necesita un programa cargador para cargar los programas y aplicaciones en memoria. El cargador, como el intérprete de tarjetas tienen que hacer E/S. Por lo tanto, el monitor también tiene una serie de manejadores de dispositivos para los dispositivos de E/S del sistema. Para evitar lecturas erróneas, solo el monitor podrá acceder físicamente a la información. Por este motivo surge la necesidad de la existencia de instrucciones privilegiadas (Modo Maestro/Esclavo).

11.4. BATCH SOFISTICADO (PERFORMANCE)

La aparición de sistemas batch mejoran la utilización de los sistemas, y mejoran la performance y el rendimiento. A pesar que el secuenciamiento automático elimina la manipulación humana, que es muy lenta, la CPU sigue estando poco utilizada. El problema reside en los dispositivos de E/S, que al ser mecánicos son mucho más lentos que la CPU.

Por ejemplo, un assembler puede procesar 300 o más tarjetas por segundo, mientras que una lectora muy veloz puede leer solo 1200 tarjetas por minuto. Entonces, para ensamblar un programa de 1579 tarjetas, se tardan 4.8 segundos de ensamblado, mientras que 78.9 segundos se ocupan en leer tarjetas. El procesador esperó 74.1 segundos, o sea, un desperdicio del 93.9%.

El mismo problema ocurre para las operaciones de salida. El problema es que mientras ocurre una operación de E/S, la CPU está desocupada esperando que termine la E/S, y mientras la CPU está ejecutando, los dispositivos de E/S están libres.

11.4.1. Operación off-line

Una solución para el problema anterior fue reemplazar las lectoras de tarjetas e impresoras, con unidades de cinta magnética. En vez que la CPU leyera directamente de las tarjetas, las mismas primero se copiaban en una cinta magnética. Cuando ésta estaba suficientemente llena, se llevaba a la computadora. Cuando un programa quería leer una tarjeta, lee la cinta. Similarmente, la salida se hace en una cinta para imprimirse posteriormente. Las lectoras e impresoras operaban off-line, no en la computadora principal.

La ventaja principal es que la CPU se libera de la espera de lectura de tarjetas, que son muy lentas, y la misma se hace por medio de cintas, que son mucho más rápidas. Además no son necesarios cambios en los programas de aplicación: los mismos llaman al driver de la lectora de tarjetas, que se reemplaza por un driver de lectura de cinta magnética. De esta forma, los programas de aplicación no deben ser cambiados: solo el driver. Los programas usan dispositivos de E/S lógicos.

La ventaja real en la operación off-line es la posibilidad de usar múltiples lectoras-escritoras magnéticas para una sola CPU. Si la CPU puede procesar entrada a dos veces la velocidad de la cinta, entonces dos lectoras trabajando simultáneamente pueden producir suficiente cinta para mantener a la CPU ocupada.

11.4.2. Buffering

Esta es otra solución a la lentitud de los dispositivos de E/S. La idea es muy simple: después que se leyó un dato, y la CPU está operando en el mismo, el dispositivo de entrada comienza a leer el próximo dato inmediatamente. La CPU y el dispositivo están ocupados simultáneamente. Un buffering similar se puede hacer para la salida de datos.

La unidad natural de datos es el registro. Puede ser un registro físico (una línea de impresión, un caracter del teclado, o un bloque de una cinta), o un registro lógico (una línea de entrada, una palabra, o un arreglo). Los

registros lógicos están definidos por la aplicación; los físicos, por la naturaleza del dispositivo de E/S. Los registros son las unidades de datos usados para el buffering.

En la práctica, es difícil que el buffering pueda mantener ocupados simultáneamente a la CPU y a los dispositivos de E/S. Si la CPU esta trabajando en un registro, mientras que un dispositivo está trabajando con otro, o la CPU o el dispositivo pueden terminar primero.

Si la CPU termina primero, tiene que esperar, porque no puede procesar otro registro hasta que el dispositivo no haya terminado con el anterior. Si el dispositivo termina primero, o espera o puede proceder a leer otro registro. Los buffers que contienen registros que ya han sido leídos y no procesados, generalmente se usan para poder mantener varios registros.

El problema principal del buffering es detectar que terminó una E/S tan pronto como sea posible, para poder lanzar la próxima. La solución para este problema son las interrupciones. Cuando un dispositivo de E/S termina con una operación, interrumpe a la CPU. Esta para de hacer lo que este haciendo, y se hace una transferencia de control a la rutina de atención de la interrupción. Esta rutina chequea si el buffer no esta lleno (para un dispositivo de entrada) o vacío (para uno de salida), y lanza el próximo pedido de E/S. Entonces, la CPU resume el cálculo interrumpido. De esta forma, los dispositivos de E/S y la CPU operan a máxima velocidad.

El buffering afecta la performance suavizando las diferencias en las variaciones de tiempo que toma procesar un registro. Si, en promedio, las velocidades de CPU y de dispositivos de E/S son similares, el buffering permite que ambos procesen casi a velocidad máxima.

Sin embargo, si la CPU es mucho más rápida que un dispositivo, el buffering no afecta demasiado la performance, porque la CPU tendrá que esperar que el dispositivo se libere aún cuando todos los buffers disponibles estén llenos.

Esta situación también ocurre con trabajos limitados por E/S (I/O-bound), donde la cantidad de E/S en relación con los cálculos es muy grande. Como la CPU es más rápida que los dispositivos, la velocidad de ejecución esta limitada por la velocidad del dispositivo de E/S. Si, por otro lado, en el caso de trabajos limitados por CPU (CPU bound), donde la cantidad de cálculos es tan grande que los buffers de entrada están siempre llenos, y los de salida vacíos, la CPU no puede emparejarse con los dispositivos de E/S.

11.4.3. Spooling

Esta es la mejor solución para todos los problemas mencionados anteriormente.

En muchos sistemas se comenzó a reemplazar los sistemas off-line y a utilizarse discos, al estar estos más disponibles. Los mismos mejoraron la operación off-line, al ser más rápidos que las cintas. El problema con éstas es que no se puede escribir en un extremo de la misma mientras la CPU lee del otro. Los discos eliminan este problema: moviendo la cabeza de un área del disco a la otra, sin el problema del rebobinado, el disco puede cambiar del área de tarjetas que se están usando, para almacenar nuevas tarjetas en otra posición.

Basado en esta ventaja se crean los sistemas de SPOOL (Simultaneous Peripheral Operation On-Line). Se leen las tarjetas directamente desde la lectora en el disco. La ubicación de la imagen de las tarjetas en el disco se almacena en una tabla del sistema operativo. Cada trabajo se almacena en la tabla. Cuando se ejecuta un trabajo, sus pedidos para la lectora de tarjetas se satisfacen leyendo su imagen desde el disco. De la misma forma, cuando se pide la impresora, esa línea se copia en el disco. Cuando el trabajo termina, se imprime la salida completa.

El buffering superpone la E/S de un trabajo con sus propios cálculos. La ventaja del spooling es que superpone la E/S de un trabajo con los cálculos de otros trabajos. Esto tiene un efecto directo sobre la performance: la CPU y los dispositivos de E/S se mantienen ocupados con tasas muy altas, particularmente si hay una mezcla de trabajos CPU-bound e I/O-bound.

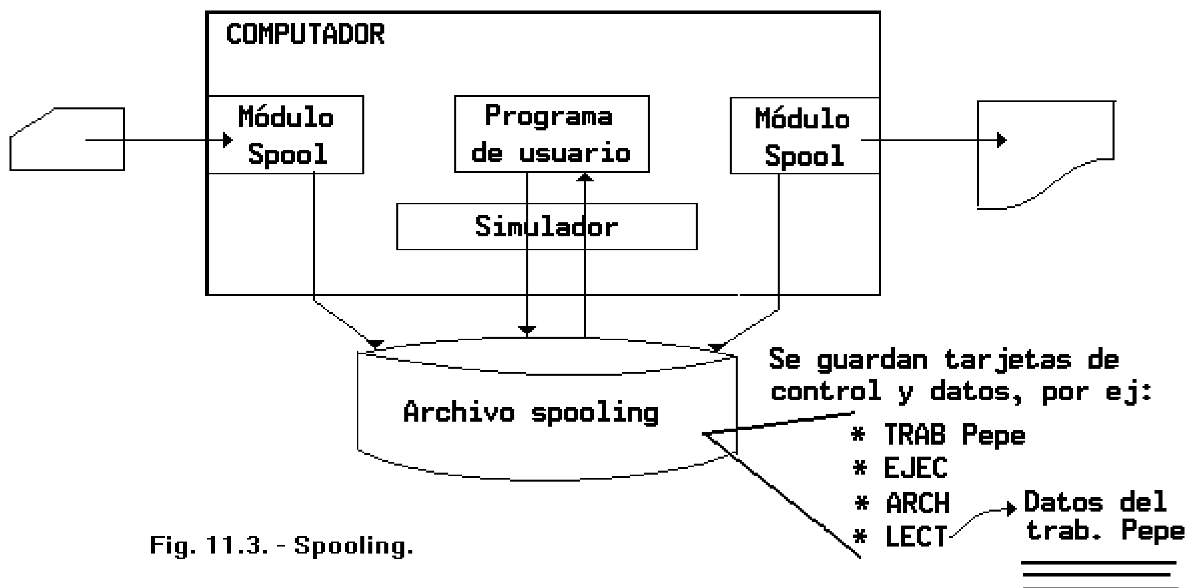


Fig. 11.3. - Spooling.

Además, el spooling provee una estructura de datos importante: una cola de trabajos. El spooling resulta en varios trabajos que han sido leídos y esperan en el disco, listos para ser ejecutados. Esta cola permite al sistema operativo elegir qué trabajo ejecutar luego, para aumentar la utilización de la CPU. Si los trabajos vienen directamente en tarjetas, o aún en una cinta, no es posible ejecutar trabajos en distinto orden al de llegada. En cambio, al tener los trabajos en un dispositivo de acceso directo surge la posibilidad de hacer una Planificación de Trabajos. Esta selección de programas lleva a la multiprogramación, o sea, tener más de un programa cargado en memoria.

11.5. MULTIPROGRAMACION

Es un intento de aumentar la utilización de la CPU, logrando que ésta siempre tenga algo que ejecutar. La idea es la siguiente: el SO levanta uno de los trabajos de la cola de trabajos y comienza a ejecutarlo. En el momento en que tenga que esperar por algo (montar una cinta, escribir algo en el teclado, terminar una operación de E/S), la CPU se entrega a otro trabajo, y así sucesivamente. En un sistema monoprogramado, la CPU hubiera estado desocupada, a pesar de que hay trabajo que hacer. Prácticamente por este tema es que surgen todos los administradores a estudiar, y lo que hace (de acuerdo a sus algoritmos), la mayor o menor eficiencia de un SO.

11.6. TIME SHARING

Los primeros sistemas batch consisten en el loteo de trabajos similares. Las tarjetas y cintas solo permiten acceso secuencial a los programas y datos, y solo se puede usar una aplicación a la vez.

Cuando aparecen los discos, se puede acceder simultáneamente a todas las aplicaciones. Ahora, los sistemas batch no están más definidos por el loteo de trabajos similares, sino por otras características. *La característica principal de un sistema batch es la falta de interacción entre el usuario y el trabajo mientras está ejecutando. El trabajo se prepara y un tiempo más tarde, aparece la salida.*

El tiempo entre la subsumisión del trabajo y su terminación, llamado el tiempo de turnaround, puede ser el resultado del tiempo de ejecución, o de las demoras antes que el sistema operativo comience a procesar el trabajo.

Como los usuarios no pueden interactuar con sus trabajos mientras están ejecutando, deben colocar tarjetas de control para manejar todos los resultados posibles. En un trabajo multietapa, el resultado puede depender de los resultados anteriores. Puede ser difícil definir que hacer en todos los casos. Otra desventaja es que los programas deben corregirse en forma estática, por medio de dumps de memoria. Un programador no puede modificar un programa a medida que se ejecuta.

Un sistema Interactivo o Hands-on provee comunicación on-line entre el usuario y el sistema. El usuario da instrucciones al sistema operativo o a un programa directamente, y recibe una respuesta inmediatamente, por medio de una terminal. Cuando el sistema operativo termina la ejecución de un comando, pide la próxima "tarjeta de control", no de una lectora de tarjetas sino del teclado. El usuario ejecuta un comando, espera la respuesta, y decide cuál será el próximo comando, basado en el resultado del anterior.

Los sistemas batch son apropiados para ejecutar trabajos grandes, que no tienen interacción. Los trabajos interactivos tienden a estar compuestos por muchas acciones cortas, donde los resultados de un comando puede ser impredecibles. Como el usuario se queda esperando el resultado, el tiempo de respuesta tiene que ser muy corto. Un sistema interactivo esta caracterizado por el deseo de un tiempo de respuesta corto.

Como vimos, los primeros sistemas eran interactivos, pero se perdía mucho tiempo de CPU. Los sistemas de Tiempo Compartido son el resultado de tratar de obtener un sistema interactivo a un costo razonable. Estos sistemas usan multiprogramación y planificación de CPU, para que cada usuario tenga una parte pequeña del tiempo de la computadora. Cada usuario tiene un programa separado en memoria.

Un sistema de tiempo compartido permite a los usuarios compartir simultáneamente la computadora. Como cada acción en un sistema de tiempo compartido tiende a ser corta, solo hace falta un tiempo corto de CPU para cada usuario. Como el sistema cambia rápidamente de un usuario al otro, los usuarios tienen la impresión que cada uno tiene su computadora propia.

11.7. SISTEMAS DE TIEMPO REAL

Un sistema de tiempo real generalmente se usa como un dispositivo de control en una aplicación dedicada. Hay sensores que dan datos a la computadora, que los analiza y puede ajustar controles para modificar las entradas del sensor.

Se utilizan para sistemas médicos, controles industriales, controles de experimentos científicos, etc. La característica mas importante de estos sistemas es que tienen restricciones de tiempo bien definidas, y el procesamiento tiene que hacerse dentro de ese tiempo.

11.8. MULTIPROCESAMIENTO

Son sistemas con mas de una CPU, compartiendo memoria y periféricos. Se usan dos aproximaciones.

La más común es asignar a cada procesador una tarea específica. Un procesador central controla el sistema, y los demás tienen tareas específicas, o le piden instrucciones al procesador principal. Este esquema define una relación maestro/esclavo. Ejemplos de estos sistemas son aquellos que usan un Procesador Frontal (Front-

End Processor) para manejar lectoras e impresoras a alguna distancia del procesador central. Estos sistemas están compuestos generalmente de un computador grande, que es la computadora principal (Host o Mainframe), y un computador más pequeño que es responsable de la E/S de la terminal.

El otro tipo común de multiprocesamiento es el uso de redes. En éstas, varias computadoras independientes pueden comunicarse, intercambiar archivos e información. Cada computador tiene su propio sistema operativo, y opera independientemente.

Actualmente existen en etapa experimental Sistemas Operativos distribuidos, o sea, con características similares a la anterior, pero en donde sus funciones están distribuidas entre distintos procesadores.

11.9. SERVICIOS QUE BRINDAN LOS SISTEMAS OPERATIVOS

Un sistema operativo provee ciertos servicios a los programas y a los usuarios. Estos servicios difieren de un sistema a otro, pero hay cierta clase de servicios que pueden identificarse:

- Ejecución de programas: El SO debe ser capaz de cargarlos, darles el control y determinar su fin normal o anormal.
- Operaciones de E/S: El programa del usuario no puede ejecutar operaciones de E/S directamente, por lo tanto el sistema operativo tiene que proveer ciertos medios para hacerlo.
- Manipulación del Sistema de Archivos: Los programas quieren leer y escribir archivos. Esto se hace por medio del sistema de archivos.
- Detección de errores: Estos pueden ocurrir en la CPU y memoria, en dispositivos de E/S, o en el programa del usuario. Por cada tipo de error, el sistema operativo tiene que tomar un tipo de acción distinto.
- Administración de recursos: Cuando hay varios usuarios ejecutando al mismo tiempo, hay que darles recursos a cada uno de ellos. El SO maneja distintos tipos de recursos: CPU, memoria principal, archivos, dispositivos.
- Accounting: Se quiere contabilizar qué recursos son usados por cada usuario. Estos datos, usados con fines estadísticos para configurar el sistema, pueden mejorar los servicios de computación. También se utilizan estos datos para cobrar a cada usuario los recursos utilizados.
- Protección: Los dueños de la información la quieren controlar. Cuando se ejecutan distintos trabajos simultáneamente, uno no debe poder interferir con los otros. Las demandas conflictivas para determinados recursos, deben ser administradas razonablemente.

Los servicios de los sistemas operativos son provistos de distintas formas. Dos métodos básicos son las Llamadas al Sistema y los Programas del Sistema.

El nivel más fundamental de los servicios se maneja por medio del uso de llamadas al supervisor. Estas proveen la interfaz entre un programa en ejecución y el sistema operativo. Generalmente están disponibles como instrucciones del lenguaje ensamblador.

Podemos reconocer, básicamente, los siguientes tipos de llamadas al supervisor:

- Control de Procesos
  - . Fin, Dump.
  - . Carga de otro programa (Load).
  - . Crear procesos, Terminar proceso.
  - . Esperar un tiempo.
  - . Esperar por un evento o una señal.
- Manipulación de Archivos
  - . Crear, borrar archivos.
  - . Apertura y cierre de archivos.
  - . Lectura y escritura.
- Manipulación de Dispositivos
  - . Pedir un dispositivo, liberar un dispositivo.
  - . Leer, escribir.
- Mantenimiento de Información
  - . Fecha, Hora.
  - . Atributos de Procesos, Archivos, o Dispositivos

Hay distintas formas de implementar las llamadas al supervisor dependiendo de la computadora en uso. Puede hacerse como interrupciones o como llamadas a un servidor (server). Es necesario identificar la llamada al sistema que se quiere hacer, y otro tipo de informaciones. Por ejemplo, para leer un registro, hace falta especificar el dispositivo o archivo a usar, y la dirección y longitud de memoria donde copiar la registro.

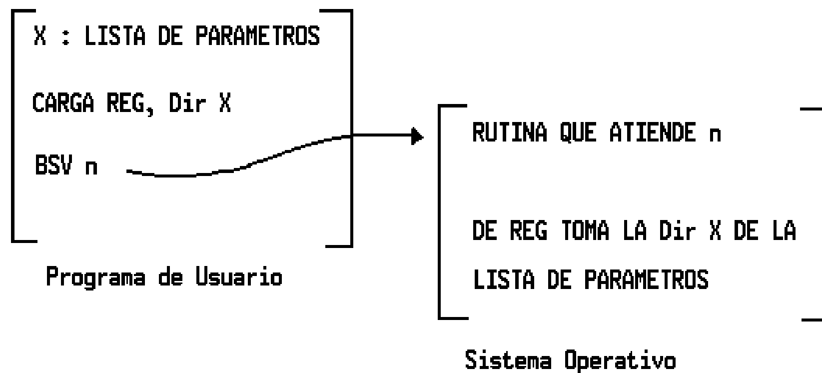


Fig. 11.4. - Pasaje de parámetros al Sistema Operativo.

Hay dos métodos generales para pasar parámetros al sistema operativo. La aproximación más simple es pasar los parámetros en registros. Sin embargo, en algunos casos hay mas parámetros que registros. En estos casos, los parámetros se almacenan en un bloque o una tabla en la memoria, y se pasa la dirección de ese bloque.

Otro aspecto de un sistema moderno es una colección de programas del sistema. Además del código del monitor residente, la mayoría de los sistemas proveen una gran cantidad de programas del sistema para resolver problemas comunes y otorgar un entorno más conveniente para la construcción y ejecución del programa.

Los programas del sistema se dividen en varias categorías:

- Manipulación de archivos: crear, copiar, renombrar, borrar, imprimir, listar, manipular, y hacer vuelcos de archivos y directorios.
- Información de estado: Hay programas para pedir fecha, hora, cantidad de memoria o disco, numero de usuarios, etc.
- Modificación de archivos: editores de texto para crear y modificar los contenidos de archivos.
- Soporte para lenguajes de programación: compiladores, assemblers, intérpretes de los lenguajes de programación más comunes.
- Carga y ejecución de programas: cargadores absolutos, cargadores reubicables, linkeditores y sistemas de debugging.
- Programas de aplicación.

11.10. **ESTRUCTURA DE SISTEMAS OPERATIVOS**

El sistema operativo se divide lógicamente en pequeños módulos y se crea una interfase bien definida para estos módulos.
Cada uno de estos módulos o piezas deben tener su función, sus inputs y outputs cuidadosamente definidos.

El sistema operativos DOS no cuenta con una buena división de estos módulos ya que no se encuentra bien particionado permitiendo el acceso directo de los programas de aplicación a rutinas básicas de E/S para grabar directamente en el display o en los discos, por ello el sistema es vulnerable a estos programas los que pueden provocar el crash del sistema.

La estructura de este sistema puede verse en la figura 11.5.

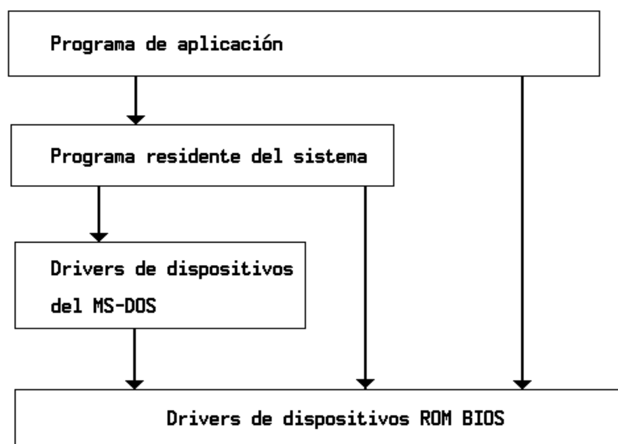


Fig. 11.5.

Otro ejemplo de un sistema operativo que no fue bien construido lo constituye el UNIX. Este sistema se encuentra divido en dos partes una de ellas comprende los programas del sistema y la otra el kernel. El kernel se encuentra divido en drivers de dispositivos y en las interfases (ver figura 11.6).

Lamentablemente este kernel combina demasiada funcionalidad en un solo nivel.

Las llamadas al sistema definen la interfase del programador al UNIX. El conjunto de programas de siste-

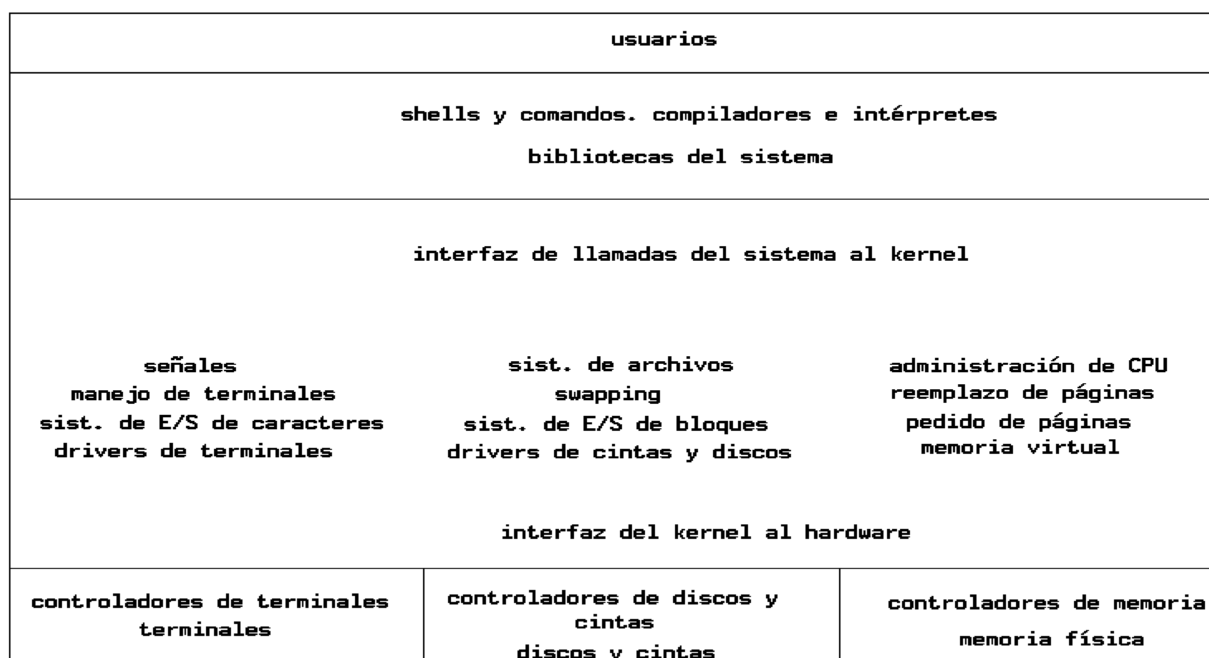


Fig. 11.6.

ma usualmente disponibles definen la interfase del usuario. Ambas definen el contexto al cual el kernel debe dar soporte.

Otras mejoras posteriores al UNIX han separado el kernel en más módulos, por ejemplo en el sistema operativo AIX de IBM se lo dividió en dos partes. El MACH de Carnegie-Mellon redujo el kernel a un conjunto pequeño de funciones básicas trasladando todo lo no esencial a los niveles del sistema o incluso al del usuario.

11.10.1 Diseño en Capas

Para mejorar casos como el UNIX se recurre a un diseño en capas, en donde cada capa se construye sobre la anterior. La capa 0 es el hardware y la capa N es la interfase de usuario.

Una capa (por ejemplo la capa M) consiste en algunas estructuras de datos y un conjunto de rutinas que operan sobre esos datos y que pueden ser invocadas por capas o niveles superiores. La capa M puede invocar operaciones en capas inferiores.

La principal ventaja de una estructura en capas es la modularidad. Las capas se seleccionan de tal manera que cada una utiliza funciones (operaciones) y servicios solo de capas de nivel inferior.

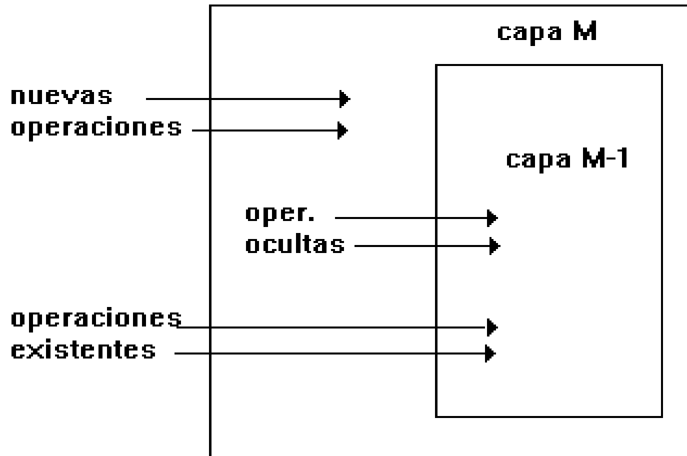


Fig. 11.7. - Un sistema operativo en capas.

Este esquema facilita la depuración aislada de cada capa. El primer nivel puede ser depurado sin tener en cuenta el resto del sistema ya que éste utiliza el hardware básico. Una vez depurado este nivel se puede proceder con el siguiente y así siguiendo.

Una capa no necesita saber cómo se implementan las operaciones de las capas inferiores sino que le basta con saber qué le brindan estas operaciones.

El esquema de capas fue originalmente utilizado en el sistema operativo THE (Technische Hogeschool Eindhoven). Este sistema se definió en 6 capas, de las cuales la capa inferior era el hardware, la siguiente implementaba la administración de la CPU, la siguiente el manejo de la memoria (memoria virtual), la tercera capa contenía los drivers para la consola del operador, y en la cuarta se encuentra el buffering de E/S pensado de esta forma para permitir que los buffers se pudieran utilizar en la memoria virtual y además para que los errores de E/S pudieran ser vistos en la consola del operador.

Nivel 5 : programas del usuario
Nivel 4 : buffering de E/S
Nivel 3 : driver de consola
Nivel 2 : manejo de memoria
Nivel 1 : manejo de CPU
Nivel 0 : hardware

Fig. 11.8. Estructura del THE.

Por ejemplo, en el sistema VENUS diseñado también en capas los niveles bajos (de 0 a 4) conciernen con las administración de la CPU y la memoria.

Nivel 6 : programas de usuario
Nivel 5 : drivers de disp. y mem.
Nivel 4 : memoria virtual
Nivel 3 : canal de E/S
Nivel 2 : administración de CPU
Nivel 1 : intérprete de instrucc.
Nivel 0 : hardware

Fig. 11.9. - Estructura del VENUS.

La mayor dificultad en el diseño de un sistema en capas radica en definir las capas o niveles ya que un nivel solo puede utilizar niveles inferiores, por lo tanto es necesaria una cuidadosa planificación.

Por ejemplo el driver para manejar los discos de la memoria virtual deberían estar en un nivel inferior a las rutinas de la memoria virtual en sí ya que estas últimas requieren del uso de este manejador.

Problemas de esta índole han provocado un retroceso en el diseño en capas en los últimos años. Se han diseñado menos capas con más funcionalidad proveyendo la mayoría de las ventajas del código modular y evitando los problemas de la definición e interacción de las capas.

El OS/2 un descendiente directo del MS-DOS fue creado para superar las limitaciones de este último. El OS/2 provee multitasking y operación en modo dual como así también otras nuevas mejoras.

En contraposición a la estructura del MS-DOS la estructura del OS/2 se encuentra diseñada en capas que por ejemplo, no permiten el acceso directo del usuario a facilidades de bajo nivel lo que otorga un mayor control al sistema operativo sobre el hardware y un mayor conocimiento de qué recursos está utilizando cada programa de usuario.

En la Figura 11.10 podemos ver la estructura en capas del sistema operativo OS/2.

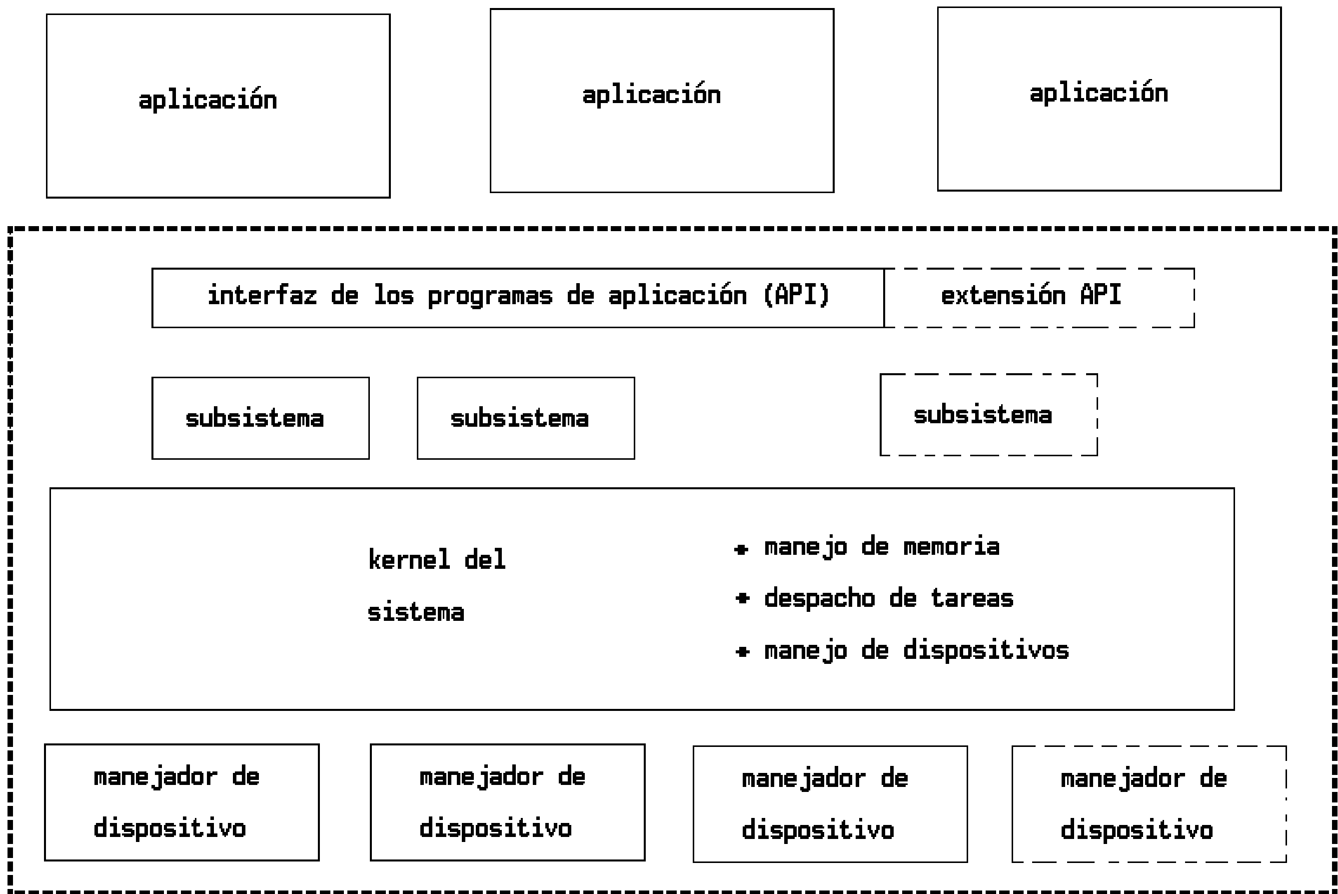


Fig. 11.10. - Estructura en capas del OS/2.

APENDICE I

RESUMEN DE INTERRUPCIONES

Enumeramos a continuación los diferentes tipos de interrupciones existentes, a saber :
- Externas (Consola del operador)
- Fin de E/S.
- Llamadas al SO:
  . Fin (normal o anormal).
  . E/S (en espera, o sin - sincronización -)
  . Información

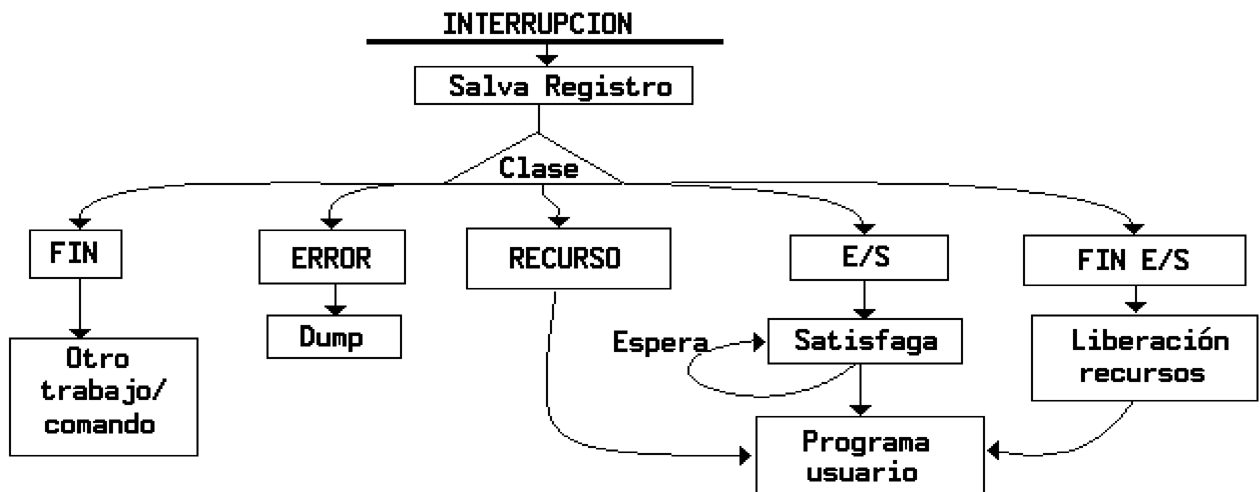


Fig. 11.11. - Diagrama de atención de interrupciones de un solo nivel.

- Errores.
- Reloj de intervalos.

Si tuviésemos un solo nivel de atención de interrupciones, un diagrama sencillo, podría ser el de la Fig. 11.11.

Tener varios niveles de atención de interrupciones da la posibilidad de trabajar en modalidad multitarea.

A lo largo del curso veremos cada uno de los administradores necesarios, y sus distintos tipos y algoritmos posibles para hacer eficiente un SO.

La idea final de la parte de Sistemas Operativos es conocer el flujo de un programa a lo largo de toda su ejecución, sus estados, y las tablas y programas que intervienen en la vida del programa/trabajo/comando.

Damos como adelanto el gráfico de la Figura 11.12.

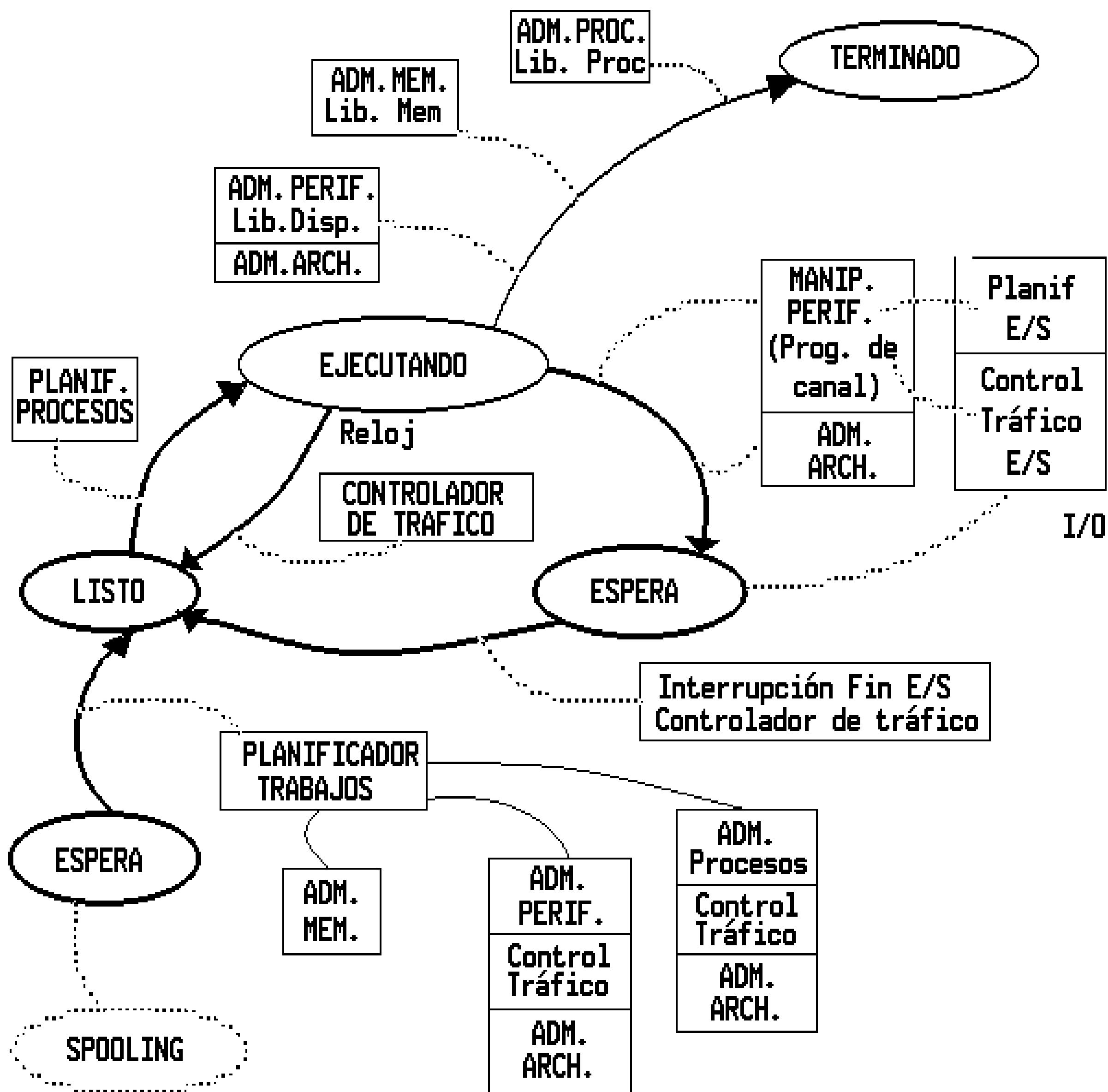


Fig. 11.12. - Un típico Diagrama de Transición de Estados.

El esquema general del Sistema Operativo que veremos tendrá una descripción en capas como el de la Fig. 11.13.

Expliquemos brevemente este gráfico:

- Hardware: Set de instrucciones + extracódigos (SVC).
- Núcleo: Implementa Procesos, su comunicación (como semáforos), y administración y sincronización del procesador.
- Administrador de Memoria: Provee servicios de administración de memoria.
- Administrador de Periféricos: Provee servicios de E/S física y administra el uso de los periféricos.
- Administrador de Archivos: Administra el uso y acceso a archivos y provee las abstracciones de registro lógico y archivo.
- Planificador de Trabajos: Administra los recursos de forma global.

- Shell: Maneja la comunicación con el usuario.

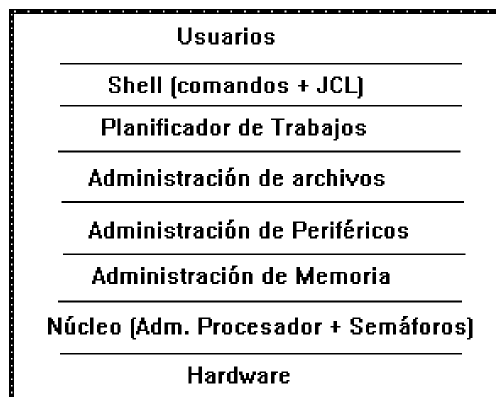


Fig. 11.13. - Diseño en capas de un Sistema Operativo.

CAPITULO 12

ADMINISTRACIÓN DEL PROCESADOR

12.1. – THREADS - Introducción

En la mayoría de los sistemas operativos tradicionales cada proceso tiene un espacio de direcciones y un hilo de control. De hecho, ésa es casi la definición de un proceso. Consideremos, por ejemplo, un servidor de archivos que debe bloquearse en forma ocasional en espera de acceso al disco. Si el servidor tiene varios hilos de control podría ejecutarse un segundo hilo mientras el primero duerme. Esto no es posible si se crean dos procesos servidores independientes ya que deben compartir un buffer en común, lo que implicaría que deben estar en el mismo espacio de direcciones.

Un Thread (hilo), a veces llamado un lightweight process, es una unidad básica de uso de CPU, y cada hilo posee un Program Counter, un Register Set y un Stack. En muchos sentidos los hilos son como pequeños mini-procesos. Cada thread se ejecuta en forma estrictamente secuencial compartiendo la CPU de la misma forma que lo hacen los procesos, solo en un multiprocesador se pueden realizan en paralelo.

Los hilos pueden crear hilos hijos y se pueden bloquear en espera de llamadas al sistema, al igual que los procesos regulares. Mientras un hilo está bloqueado se puede ejecutar otro hilo del mismo proceso.

Puesto que cada hilo tiene acceso a cada dirección virtual (comparten un mismo espacio de direccionamiento), un hilo puede leer, escribir o limpiar la pila de otro hilo. No existe protección entre los hilos debido a que es imposible y no es necesario, ya que no son hostiles entre sí, es más a veces cooperan. Aparte del espacio de direcciones, comparten el mismo conjunto de archivos abiertos, procesos hijos, cronómetro, señales, etc.

ELEMENTOS POR HILOS	ELEMENTOS POR PROCESOS
Contador del programa	Espacio de dirección
Pila	Variables globales
Conjunto de Registros	Archivos Abiertos
Hilos hijos	Procesos hijos
Estado	Cronómetros
	Señales
	Semáforos
	Información contable

12.2. - USO DE LOS HILOS

Los hilos se inventaron para permitir la combinación del paralelismo con la ejecución secuencial y el bloqueo de las llamadas al sistema. Existen 3 formas de organizar un proceso de muchos hilos en un server.

Estructura Servidor Trabajador
Estructura En Equipo
Estructura Entubamiento

12.2.1. - Estructura Servidor Trabajador

Existe un hilo en el servidor que lee las solicitudes de trabajo en un buzón del sistema, examina éstas y elige a un hilo trabajador inactivo y le envía la solicitud, la cual se realiza con frecuencia al colocarle un puntero al mensaje en una palabra especial asociada a cada hilo. El servidor despierta entonces al trabajador dormido (un signal al semáforo asociado).

El hilo verifica entonces si puede satisfacer la solicitud desde el bloque cache compartido, sino puede inicia la operación correspondiente (por ejemplo podría lanzar una lectura al disco) y se duerme nuevamente a la espera de la conclusión de ésta. Entonces, se llama al planificador para iniciar otro hilo, ya sea hilo servidor o trabajador.

Otra posibilidad para esta estructura es que opere como un hilo único. Este esquema tendría el problema de

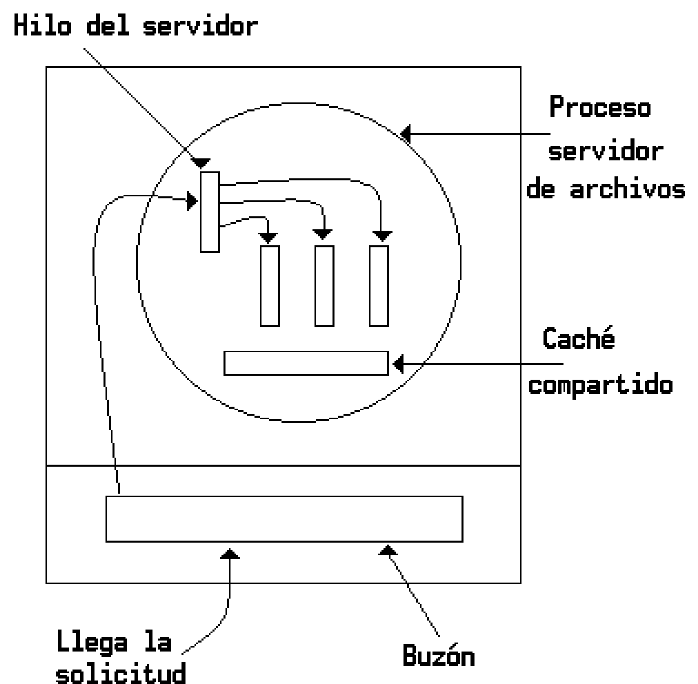


Fig. 12.1.- Hilo trabajador.

que el hilo del servidor lanza un pedido y debe esperar hasta que éste se complete para lanzar el próximo (secuencial).

Una tercera posibilidad es ejecutar al servidor como una gran máquina de estado finito. Esta trata de no bloquear al único hilo mediante un registro del estado de la solicitud actual en una tabla, luego obtiene de ella el siguiente mensaje, que puede ser una solicitud de nuevo trabajo, o una respuesta de una operación anterior. En el caso del nuevo trabajo, éste comienza, en el otro caso se busca la información relevante en la tabla y se procesa la respuesta. Este modelo no puede ser usado en RPC puesto que las llamadas al sistema no son bloqueantes, es decir no se permite enviar un mensaje y bloquearse en espera de una respuesta (como veremos más adelante).

En este esquema hay que guardar en forma explícita el estado del cómputo y restaurarlo en la tabla para cada mensaje enviado y recibido. Este método mejora el desempeño a través del paralelismo pero utiliza llamadas no bloqueantes y por lo tanto es difícil de programar.

Modelo	**Características**
Hilos	Paralelismo; llamadas al sistema bloqueantes
Servidor de un solo hilo; Sin paralelismo	llamadas al sistema bloqueantes
Máquina de estado finito; Paralelismo	llamadas al sistema no bloqueantes

12.2.2. - **Estructura en Equipo**

Todos los hilos son iguales y cada uno obtiene y procesa sus propias solicitudes. Cuando un hilo no puede manejar un trabajo por ser hilos especializados se puede utilizar una cola de trabajo. Esto implica que cada hilo verifique primero la cola de trabajo antes de mirar el buzón del sistema.

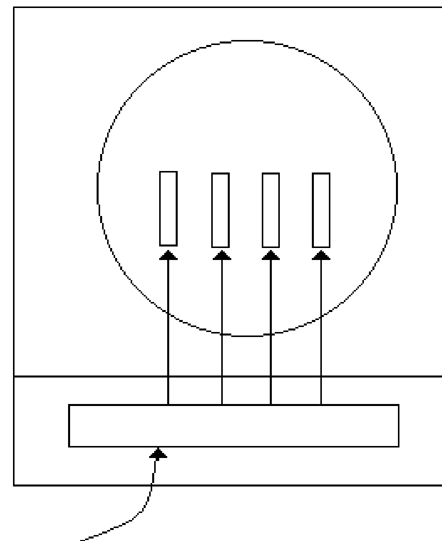


Fig.12.2. - Estructura en equipo.

12.2.3. - **Estructura de Entubamiento** (pipeline)

El primer hilo genera ciertos datos y los transfiere al siguiente para su procesamiento. Los datos pasan de hilo en hilo y en cada etapa se lleva a cabo cierto procesamiento. Esta puede ser una buena opción para productor/consumidor, no así para los servidores de archivos.

12.2.4. - **Otros usos de los hilos**

En un cliente los threads son usados para copiar un archivo a varios servidores, por ejemplo si un cliente quiere copiar un mismo archivo a varios servidores puede dedicar un hilo para que se comunique con cada uno de ellos.

Otro uso de los hilos es el manejar las señales, como las interrupciones del teclado. Por ejemplo se dedica un hilo a esta tarea que permanece dormido y cuando se produce una interrupción el hilo se despierta y la procesa.

Existen aplicaciones que se prestan para ser programadas con el modelo de hilos, por ejemplo el problema de los productores-consumidores. Nótese que como comparten un buffer en común no funcionaría el hecho de tenerlos en procesos ajenos.

Por último nótese la utilidad de los hilos en sistemas multiprocesadores en dónde pueden realmente ejecutarse en forma paralela.

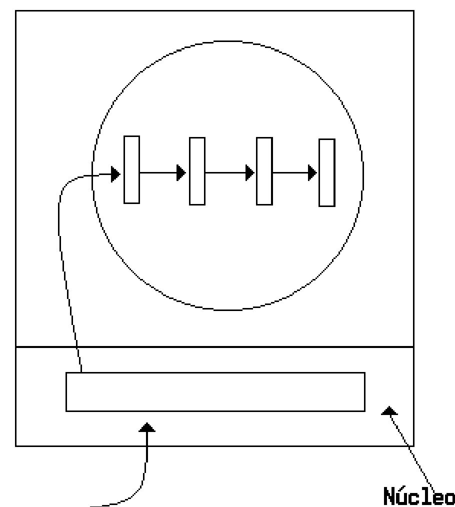


Fig. 12.3.- Estructura pipeline.

12.3. - **ASPECTOS DEL DISEÑO DE PAQUETES DE THREADS**

Un conjunto de primitivas relacionadas con los hilos disponibles para los usuarios se llama un *paquete de hilos*. Se pueden tener hilos estáticos o dinámicos.

En un diseño estático se elige el número de hilos al escribir el programa, o durante su compilación. Cada uno de ellos tiene una pila fija.

En el dinámico se permite la creación y destrucción de los hilos en tiempo de ejecución. La llamada para la creación de un hilo establece el programa principal del hilo (como si fuera un apuntador a un procedimiento) y un tamaño de pila, también otros parámetros, como ser prioridad de planificación.

La llamada devuelve un identificador de hilo para ser usado en posteriores llamadas relacionadas con ese hilo. Entonces un proceso se inicia con un único hilo, pero puede crear el numero necesario de ellos.

La terminación de los hilos puede ser de dos maneras, al terminar su trabajo o bien desde el exterior. En el ejemplo del servidor de archivos los hilos se crean una vez iniciado el programa y nunca se eliminan.

Ya que los hilos comparten una memoria común el acceso a datos comunes se programa mediante regiones críticas que se implementan mediante el uso de semáforos, monitores u otras construcciones similares. Se utiliza el semáforo de exclusión *mutex* sobre el cual se definen las operaciones de cierre (LOCK) y liberación (UNLOCK). Existe además una operación TRILOCK que para el cierre funciona igual que el Lock pero si el semáforo está cerrado en lugar de bloquear el hilo regresa un código de estado que indica falla.

Se encuentra disponible también la *variable de condición*, similar en todo a la que se utiliza en monitores (como veremos más adelante).

El código de un hilo consta al igual que un proceso de varios procedimientos, pudiendo tener variables locales, globales y variables del procedimiento. De éstas las globales producen problemas, ya que el valor de una variable global puesto por un hilo que se duerme puede ser modificado por otro, creando así una incoherencia cuando se despierta. Las soluciones que se presentan son:

a) Prohibir las variables globales: esto presenta conflictos con el software ya existente, como por ejemplo Unix.
b) Asignarle a cada hilo sus propias variables globales particulares. Esto introduce un nuevo nivel de visibilidad, ya que las variables son visibles a todos los procedimientos de un hilo además de las variables visibles a un procedimiento específico y las visibles a todo el programa. Esta alternativa tiene el inconveniente de no ser posible de implementar en la mayoría de los lenguajes de programación. Una forma de hacerlo es asignar un bloque de memoria a las variables globales y transferirlas a cada procedimiento como un parámetro adicional.
c) Nuevos procedimientos en bibliotecas para crear, leer y escribir estas variables. La creación de una variable global implicaría la asignación de un apuntador en un espacio de almacenamiento dedicado a ese hilo de forma que solo él tiene acceso a la variable global definida.

12.3.1 - **Llamadas.**

Presentamos un resumen de las llamadas que pueden existir para el manejo de los hilos o threads:

Llamadas de manejo de Threads
- Create, Exit, Join, Detach

Llamadas de Sincronización (Manejo de regiones criticas)
- Mutex_init, Mutex_destroy, Mutex_lock, Mutex_trylock, Mutex_unlock

Llamadas de Condición Variables (usados para el bloqueo de recursos)
- Cond_init, Cond_destroy, Cond_wait, Cond_signal, Cond_broadcast

Llamadas de Scheduling (administran las prioridades de los hilos)
- Setscheduler, Getscheduler, Setprio, Getprio

Llamadas de Eliminación
- Cancel, Setcancel

12.4. - **IMPLEMENTACIÓN DE UN PAQUETE DE HILOS**

Aquí trataremos la planificación de los hilos mediante distintos algoritmos, ya que el usuario puede especificar el algoritmo de planificación y establecer las prioridades en su caso. Existen dos formas de implementar un paquete de hilos:

1) Colocar todo el paquete de hilos en el espacio del usuario (el núcleo no conoce su existencia),
2) Colocar todo el paquete de hilos en el espacio del núcleo (el núcleo sabe de su existencia)

12.4.1. - Paquete de hilos en el espacio del usuario

La ventaja es que este modelo puede ser implantado en un sistema operativo que no tenga soporte para hilos. Por ejemplo UNIX no soporta hilos.

Los hilos se ejecutan arriba de un SISTEMA DE TIEMPO DE EJECUCIÓN (Intérprete), que se encuentra en el espacio del usuario en contacto con el núcleo. Por ello, cuando un hilo ejecuta una llamada al sistema, se duerme, ejecuta una operación en un semáforo o mutex, o bien cualquier operación que pueda provocar su suspensión. Se llama a un procedimiento del sistema de tiempo de ejecución el cual verifica si debe suspender al hilo. En caso afirmativo almacena los registros del hilo (propios) en una tabla, busca un hilo no bloqueado para ejecutarlo, vuelve a cargar los registros guardados del nuevo hilo, y apenas intercambia el apuntador a la pila y el apuntador del programa, el hilo comienza a ejecutar.

El intercambio de hilos es de una magnitud menor en tiempo que una interrupción siendo esta característica un fuerte argumento a favor de esta estructura.

Este modelo tiene una gran escalabilidad y además permite a cada proceso tener su propio algoritmo de planificación. El sistema de tiempo de ejecución mantiene la ejecución de los hilos de su propio proceso hasta que el núcleo le retira el CPU.

12.4.2. - Paquete de hilos en el núcleo

Para cada proceso, el núcleo tiene una tabla con una entrada por cada hilo, con los registros, estado, prioridades, y demás información relativa al hilo (ídem a la información en el caso anterior), solo que ahora están en el espacio del núcleo y no dentro del sistema de tiempo de ejecución del espacio del usuario.

Todas las llamadas que pueden bloquear a un hilo se implantan como llamadas al sistema, esto representa un mayor costo que el esquema anterior por el cambio de contexto. Al bloquearse un hilo, el núcleo opta entre ejecutar otro hilo listo, del mismo proceso, o un hilo de otro proceso.

12.4.3. - **PROBLEMAS**

a) Implementación de las llamadas al sistema con bloqueo.

Un hilo hace algo que provoque un bloqueo, entonces en el esquema de hilos a nivel del usuario no se puede permitir que el thread realice en realidad la llamada al sistema ya que con esto detendría todos los demás hilos y uno de los objetivos es permitir que usen llamadas bloqueantes pero evitando que este bloqueo afecte a los otros hilos.

En cambio en el esquema de hilos en el núcleo directamente se llama al núcleo el cual bloquea al hilo y luego llama a otro.

Hay una forma de solucionar el problema en el primer esquema y es verificar de antemano que una llamada va a provocar un bloqueo, en caso positivo se ejecuta otro hilo. Esto lleva a escribir parte de las rutinas de la biblioteca de llamadas al sistema, si bien es ineficiente no existen muchas alternativas. Se denomina jacket.

Algo similar ocurre con las fallas de página. Si un hilo produce una falla de página el núcleo que desconoce que dentro del proceso del usuario hay varios hilos bloquea todo el proceso.

b) Administración de los hilos (scheduling).

En el esquema de hilos dentro del proceso del usuario cuando un hilo comienza su ejecución ninguno de los demás hilos de ese proceso puede ejecutarse a menos que el primer hilo entregue voluntariamente el CPU. Dentro de un único proceso no existen interrupciones de reloj, por lo tanto no se puede planificar con quantum.

En el esquema de hilos en el núcleo las interrupciones se presentan en forma periódica obligando a la ejecución del planificador.

c) Constantes llamadas al sistema.

En el esquema de hilos a nivel usuario se necesitaría la verificación constante de la seguridad de las llamadas al sistema, es decir es mucho más simple el bloqueo en los hilos a nivel núcleo que a nivel usuario puesto que en el núcleo solo bloquea al hilo y conmuta al próximo hilo mientras que a nivel usuario necesita antes de bloquearse llamar al sistema de tiempo de ejecución para conmutar y luego bloquearse.

d) Escalabilidad

El esquema de hilos a nivel usuario tiene mejor escalabilidad ya que en el esquema de hilos a nivel núcleo éstos requieren algún espacio para sus tablas y su pila en el núcleo lo que se torna inconveniente si existen demasiados hilos.

e) Problema general de los hilos

Muchos de los procedimientos de biblioteca no son reentrantes. El manejo de las variables globales propias es dificultoso. Hay procedimientos (como la asignación de memoria) que usan tablas cruciales sin utilizar regiones críticas, pues en un ambiente con un único hilo eso no es necesario. Para poder solucionar estos problemas de manera adecuada habría que reescribir toda la biblioteca. Otra forma es proveer un jacket a cada hilo de manera que cierre un mutex global al iniciar un procedimiento. De esta forma la biblioteca se convierte en un enorme monitor.

El uso de las señales (traps - interrupciones) también producen dificultades, por ejemplo dos hilos que deseen capturar la señal de una misma tecla pero con propósitos distintos. Ya es difícil manejar las señales en ambientes con un solo hilo y en ambientes multithreads las cosas no mejoran. Las señales son un concepto típico por proceso y no por hilo, en especial, si el núcleo no está consciente de la existencia de los hilos.

12.5 – **INTRODUCCION ADMINISTRACIÓN DEL PROCESADOR**

La administración del procesador es, prácticamente, el tema central de la multiprogramación. Esta administración involucra las distintas maneras a través de las cuales el Sistema Operativo comparte el recurso procesador entre distintos procesos que están compitiendo por su uso. Esto implica directamente la multiprogramación y conlleva simultáneamente la sincronización de los mismos.

La idea de administrar el procesador eficientemente está enfocada en dos aspectos: el primero es la cantidad de procesos por unidad de tiempo que se pueden ejecutar en un sistema; y el segundo, el que importa más al usuario, es el tiempo de respuesta de esos procesos.

- Cantidad de Procesos por Unidad de Tiempo (throughput)
- Tiempo de Respuesta (turnaround time)

La idea de repartir el recurso procesador entre distintos procesos se debe a que tenemos la posibilidad de utilizar el tiempo de procesador abandonado por un proceso para que lo pueda usar otro. O sea aprovechar los tiempos muertos de un determinado proceso para que se puedan ejecutar otros.

Estos tiempos muertos se producen porque existen otras actividades que están desarrollándose sobre cierto proceso. Esas otras actividades generalmente son de E/S, y esto es posible porque existe algo que está ayudando a realizar esa E/S, es decir, existen canales o procesadores de E/S que ayudan a descargar del procesador central esa actividad.

12.6. - **Turnaround**

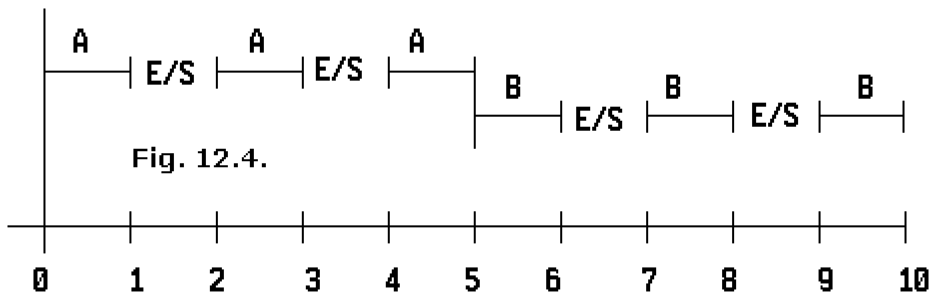


Fig. 12.4.

Normalmente nos encontramos con un proceso A que tiene una cantidad de tiempo de procesador, y otra cantidad de tiempo muerto desde el punto de vista del procesador, porque está realizando una operación de E/S, procesa nuevamente, E/S, y procesa nuevamente.

La idea es tratar de, en esos momentos en donde la actividad está descargada en un procesador especializado en E/S, usar ese tiempo para que otro proceso ejecute.

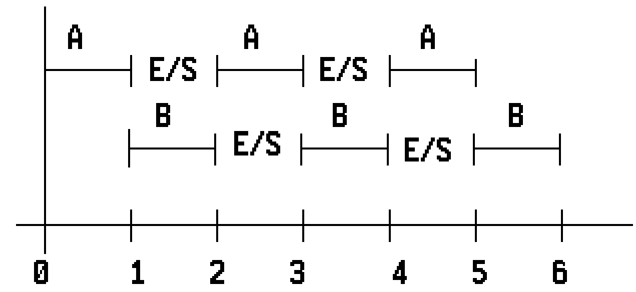


Fig. 12.5.

Supóngase que existe otro proceso homogéneo a éste, es decir, las mismas ráfagas de procesador, y los mismos tiempos de E/S (3 ráfagas de procesador y 2 operaciones de E/S), y ambos procesos se ejecutaran en monoprogramación , el primero tardaría 5 unidades, y el segundo tardará otras 5 unidades. Es decir que recién después de 10 unidades de tiempo se tendría la finalización de ambos programas, con lo cual se obtiene un tiempo de respuesta promedio de 7.5 unidades (Ver Fig. 12.4 - turnaround (10+5) / 2= 7.5 -).

Si se sumergiera esta situación en un ambiente de multiprogramación, suponiendo que los tiempos de E/S de uno de los programas coinciden con las ráfagas de procesador del otro, se podría obtener un esquema de distribución como se ve en la Fig. 12.5.

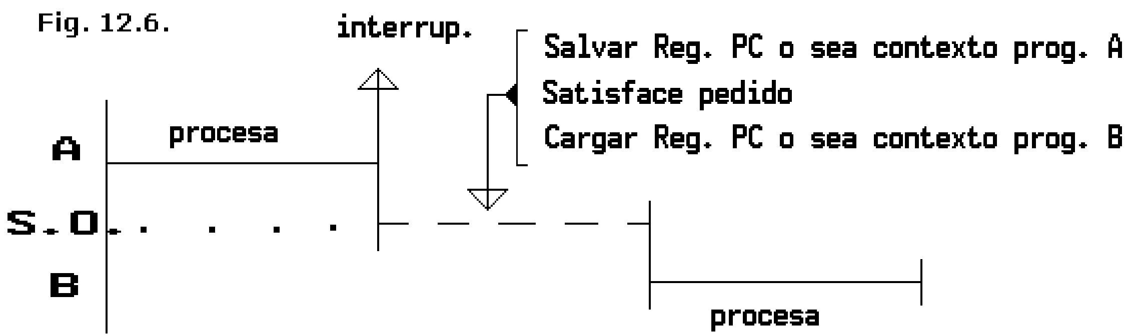
De esta manera el tiempo de respuesta promedio es de 5.5 (Turnaround) (6+5)/2= 5.5.

La ventaja es que los procesos terminan antes del valor esperado en promedio en monoprogramación. Un elemento de medida que es el tiempo medio de terminación de programa permite verificar las bondades de los distintos algoritmos que se verán a continuación.

El tiempo medio de respuesta, turnaround, se obtiene como la suma de los tiempos de terminación de los procesos dividida la cantidad de procesos. Este turnaround, en realidad, está indicando el índice de felicidad de un usuario, da un valor esperado que debe interpretarse de la siguiente manera : A las x unidades de tiempo se puede tener la esperanza de que existen procesos que ya han finalizado, por lo tanto cuánto menor es x menos se debe esperar, en promedio, para que el sistema haya dado una respuesta.

Lógicamente que suponer que las ráfagas de procesador de uno de los procesos coinciden exactamente con los tiempos de E/S del otro proceso es prácticamente una ilusión, más aún considerando que existen otros tiempos asociados a la multiprogramación que complican aún más la interacción entre los procesos. Consideremos por ejemplo el grafo de la Fig. 12.6.

Fig. 12.6.



O sea, en el primer momento se está ejecutando el proceso A, luego se produce alguna interrupción provocada por el mismo programa o recibida del exterior, se debe entonces salvar los

registros y la palabra de control del proceso A (o sea su contexto), suponiendo que fue una interrupción por una llamada al supervisor para realizar una E/S, se satisfará ese pedido, y después se cargarán los registros y la palabra de control para el proceso B (o sea su contexto).

Es decir, que los tiempos, de acuerdo a lo visto anteriormente, no van a ser tan exactos, sino que se tendrá una cantidad de tiempo para realizar las tareas antes enumeradas, y recién después de esto se podrá ejecutar el siguiente proceso.

A este tiempo dedicado a la atención de interrupciones, salvaguarda y carga de contextos, en suma, el tiempo dedicado por el sistema operativo a ejecutar sus propias rutinas para proveer una adecuada administración entre los diferentes procesos se lo suele denominar **el overhead** (sobrecarga) del Sistema Operativo.

12.7 - Tablas y Diagrama de Transición de Estados

Para poder manejar convenientemente una administración de procesador será necesario contar con un cierto juego de datos. Ese juego de datos será una tabla en la cual se reflejará en qué estado se encuentra el proceso, por ejemplo, si está ejecutando o no.

Los procesos, básicamente, se van a encontrar en tres estados :

- Ejecutando,
- Listo para la ejecución, o
- Bloqueados por alguna razón.

En base a estos estados se construye lo que se denomina **Diagrama de Transición de Estados**.

Estar en la cola de Listos significa que el único recurso que a ese proceso le está haciendo falta es el recurso procesador. O sea, una vez seleccionado de esta cola pasa al estado de Ejecución.

Se tiene una transición al estado de Bloqueados cada vez que el proceso pida algún recurso. Una vez que ese requerimiento ha sido satisfecho, el proceso pasará al estado de Listo porque ya no necesita otra cosa más que el recurso procesador.

El correspondiente Diagrama de Transición sería pues el de la Fig. 12.7.

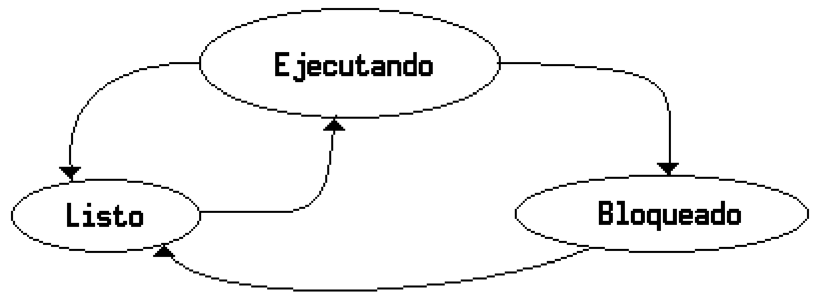


Fig. 12.7 - Diagrama de Transición de Estados.

Para manejar esa cola de listos se requiere de una tabla, y esa tabla, básicamente, lo que debe tener es una identificación de los procesos (Fig. 12.8).

Como los listos pueden ser muchos, hará falta un apuntador al primero de esa cola de listos, y posiblemente un enganche entre los siguientes en el mismo estado.

Esta tabla contiene los Bloques de Control de Procesos. En este caso se agrupan los BCP en una Tabla de Bloques de Control de Procesos (TBCP).

Fig. 12.8.

Puntero al primero de los Listos →

Id. Proceso	Estado
A	Ejec.
B	Listo
C	Bloq.
D	Listo

12.8. - Bloque de Control de Proceso (BCP)

El Bloque de Control de Procesos contiene el contexto de un proceso y todos los datos necesarios para hacer posible la ejecución de ese proceso y satisfacer sus necesidades.

Cada entrada de la tabla tiene un apuntador al bloque anterior y uno al posterior, una identificación del proceso, la palabra de control, los registros, si se está trabajando en un sistema de administración de memoria paginada un apuntador a su tabla de distribución de páginas, dispositivos que esté usando, archivos que esté usando, tiempos que hacen a la vida del proceso, el es-

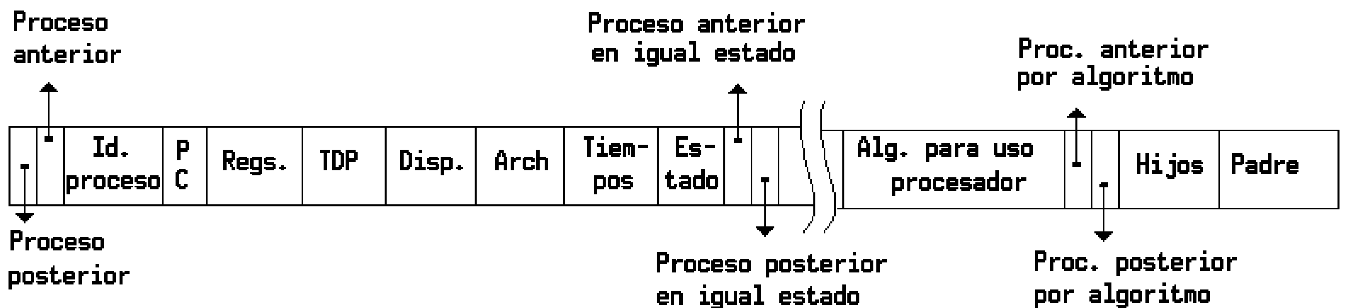


Fig. 12.9. - Contenido del Bloque de Control de Proceso (BCP).

tado, y apuntadores al anterior y posterior en el mismo estado.

Un esquema de una entrada de un Bloque de Control de Procesos puede verse en la Fig. 12.9.

Si bien es cierto que es más fácil pensar a la TBCP como una matriz, este tipo de implementación es muy rígida y rápidamente podría desembocar en que el espacio reservado para BCP's se termine.

Pensando en una implementación más dinámica se puede implementar a la TBCP como un encadenamiento de BCP's.

En forma más detallada cada uno de sus campos contiene :

- Apuntador a proceso anterior: dirección del BCP anterior (anterior en tiempo pues fue creado antes). El primer BCP tendrá una identificación que lo señale como tal y deberá ser conocida su ubicación por el Planificador de Procesos.
- Apuntador a proceso posterior: dirección del BCP posterior (posterior en tiempo, pues fue creado después). El último BCP tendrá un nil (no se descartan encadenamientos circulares, pero por ahora se los presenta como lineales).
- Identificación de Proceso: identificación única para este proceso que lo hace inconfundible con otro.
- Palabra de control: espacio reservado o apuntador en donde se guarda la PC cuando el proceso no se encuentra en ejecución.
- Registros: ídem anterior, pero para los registros de uso general del proceso.
- TDP: apuntador al lugar en donde se encuentra la Tabla de Distribución de Páginas correspondiente a este proceso. En el supuesto de tratarse de otro tipo de administración de memoria en esta ubicación se encontraría la información necesaria para conocer en qué lugar de memoria está ubicado el proceso.
- Dispositivos: apuntador a todos los dispositivos a los que tiene acceso el proceso al momento. Esta información puede ser estática si es necesario que el proceso declare antes de comenzar su ejecución los dispositivos a usar, o completamente dinámica si existe la capacidad de obtener y liberar dispositivos a medida que se ejecuta el proceso.
- Archivos: ídem Dispositivos pero para los archivos del proceso.
- Tiempos: Tiempo de CPU utilizado hasta el momento. Tiempo máximo de CPU permitido a este proceso. Tiempo que le resta de CPU a este proceso. Otros tiempos.
- Estado: Ejecutando. Listo. Bloqueado. Wait (En espera). Ocioso.
- Apuntador al BCP del proceso anterior en el mismo estado: dirección del BCP correspondiente al proceso anterior en ese mismo estado.
- Apuntador al BCP del proceso posterior en el mismo estado: ídem anterior pero al proceso posterior.
- Información para el algoritmo de adjudicación del procesador: aquí se tendrá la información necesaria de acuerdo al algoritmo en uso.
- Apuntador al BCP del proceso anterior en función del algoritmo: dependerá del algoritmo.
- Apuntador al BCP del proceso posterior en función del algoritmo: dependerá del algoritmo.
- Apuntador al BCP del Proceso Padre: dirección del BCP del proceso que generó el actual proceso.
- Apuntador a los BCP Hijos: apuntador a la lista que contiene las direcciones de los BCP hijos (generados por) de este proceso. Si no tiene contendrá nil.
- Accounting: información que servirá para contabilizar los gastos que produce este proceso (números contables, cantidad de procesos de E/S, etc.)

12.9 - **Programa y Proceso**

Un programa es una entidad pasiva, mientras que un proceso o tarea (se usan indistintamente ambos términos) es una entidad activa.

O sea que un programa es un conjunto de instrucciones, un proceso es un conjunto de instrucciones más su contexto (BCP) y en ejecución.

Qué significa realmente un proceso o una tarea dentro de un programa?. Dados dos conjuntos de instrucciones que están dentro de un programa, si esos conjuntos son completamente independientes, de manera tal que su ejecución independiente no afecta el resultado del programa, luego cada conjunto es un proceso. Por ejemplo, si se tiene el siguiente conjunto de instrucciones :

A = A + C
Z = A + 1

y por otro lado :

D = E + F
Y = D + 1

Estos dos conjuntos de instrucciones son completamente independientes, y que se ejecuten en forma independiente no afecta el resultado final del programa ya que están manejando datos distintos.

Para verlo en un caso bastante claro en el cual se podría dividir un programa en un par de tareas, supóngase que existe el siguiente conjunto de instrucciones:

READ A

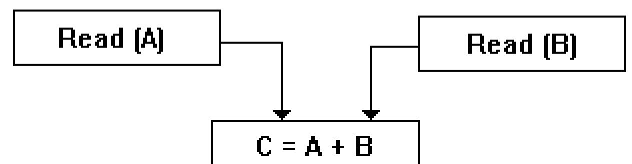


Fig. 12.10.

READ B
C = A + B

Cada instrucción READ, que desencadena una serie de operaciones, es completamente independiente de la otra que desencadena otro conjunto de instrucciones completamente independientes del otro. En cambio, la operación C = A + B es dependiente de las otras dos, desde el momento que está usando resultados que le dan las dos anteriores. O sea que este fragmento de programa podría ser descompuesto en un grafo de la manera que se ve en la Fig. 12.10.

Pudiendo descomponer de esta manera a un programa, es realmente posible que las dos operaciones, más tarde conjunto de instrucciones, pueden ser ejecutadas en forma simultánea, desde el punto de vista de la lógica del programa, es decir no afecta hasta aquí que cualquiera de las dos se ejecute antes, a partir de la asignación en sí.

Existe toda una serie de condiciones para establecer cuándo es posible hacer esto que se verán más adelante cuando se estudien los procesos concurrentes, pero hay que tener bien en claro que es posible dividir un proceso en subconjuntos de procesos que llamamos secuenciales, y esos procesos son secuenciales si están compuestos por instrucciones que dependen unas de otras.

Si el ejemplo anterior se diera en un ambiente de multiprocesamiento sería posible efectuar las dos operaciones READ en forma simultánea, algo que en cursos anteriores se veía cómo una llamada al supervisor **sin devolución** del recurso procesador al sistema operativo, porque se volvía al programa para ejecutar otra operación que era completamente independiente de la operación anterior. Esto va a requerir algún elemento de sincronización, ya que, volviendo al ejemplo anterior, no se podrá realizar la asignación hasta que no se hayan completado los READ. Esto también es tema para el futuro.

Dado un programa es posible generar, por situaciones especiales, que se divida en distintas tareas que son independientes. Y estas también compiten por el recurso procesador, luego se tendrá que administrar el procesador también para distintas tareas que componen un programa y que van a utilizar ese recurso. Con lo cual también es posible que desde un bloque de control de un programa exista un apuntador X al bloque de control de procesos correspondientes a ese programa, y que dentro de ese bloque de control de procesos, cada entrada corresponda a un proceso o tarea independiente que lo conforman.

Luego, se puede hablar de programas o de procesos en forma indistinta ya que dado un programa se puede descomponer en tareas disjuntas. En definitiva se están manejando bloques de control de procesos que van a requerir del uso del procesador.

Un programa completo puede dividirse en procesos. Esto es típico de lenguajes que permiten multitareas, como PL-1, y Pascal concurrente, ya que, o bien lo indica el programador o porque el compilador es lo suficientemente inteligente como para darse cuenta de segmentar el programa en distintos procesos, y esos procesos pueden después competir por el recurso como si fueran procesos independientes.

Utilizar este esquema dentro de una computadora que tiene un solo procesador es hacer multiprogramación . En los sistemas que tienen más de un procesador, y con la posibilidad de ejecutar cada READ en cada uno de ellos, se está en presencia de un sistema de multiprocesamiento. Más aún, si se inserta este esquema en computadoras distintas con algún mecanismo de comunicación, considerando inclusive que pueden estar ubicadas en sitios remotos, se puede decir que se está frente a un sistema de computación distribuido.

Dividir a los programas en tareas requiere luego, obviamente, de un sistema de comunicación que permita que una vez que estén divididos en tareas, se los pueda insertar en cualquier tipo de arquitectura y aprovechar entonces sus facilidades específicas.

12.10 - **Fin de un Proceso (total o temporal)**

Mientras se está ejecutando un proceso este puede abandonar su estado por diversas razones. Lo más deseable sería que el proceso terminase en forma satisfactoria.

Las causales posibles de abandono son :

- FIN NORMAL (Proceso completo)
- ERROR (Fin anormal)
- NECESITA RECURSOS (E/S, etc.) (Pasa a Bloqueado)
- DESALOJO (Por algún proceso de mayor prioridad) (Pasa a Listos)

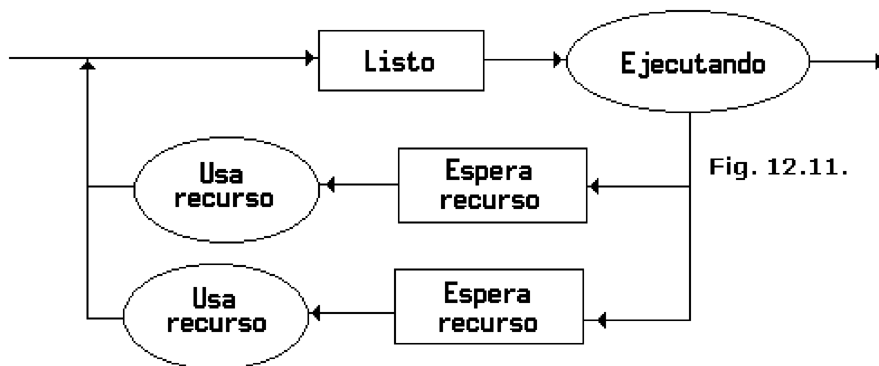


Fig. 12.11.

Las dos primeras causales de finalización se refieren al fin Total del proceso, en tanto que las dos últimas indican solamente un fin Temporal del mismo.

Desalojo significa que, por alguno de los algoritmos de administración de procesador, se considera que el tiempo de uso del procesador por parte de ese proceso ha sido demasiado alto.

Se puede graficar esta situación (Fig. 12.11) visualizando además la existencia de colas para los recursos compartidos.

12.11 - **Rutinas de Administración del Procesador**

Generalmente, y dependiendo de la literatura que se consulte, se encuentra que la administración del procesador incluye el pasaje de Retenido a Listos (la entrada del exterior a listos) como una de las partes de la administración del procesador. Lógicamente, mientras no se seleccione un proceso, va a ser imposible que éste compita por el recurso procesador.

Se denomina Planificador de Trabajos al conjunto de rutinas que realizan esta función de ingresar un proceso al sistema desde el exterior y se lo llama muy a menudo administrador de alto nivel. Es además capaz de comunicarse con el resto de los administradores para ir pidiendo los recursos que el trabajo necesitará para iniciar su ejecución.

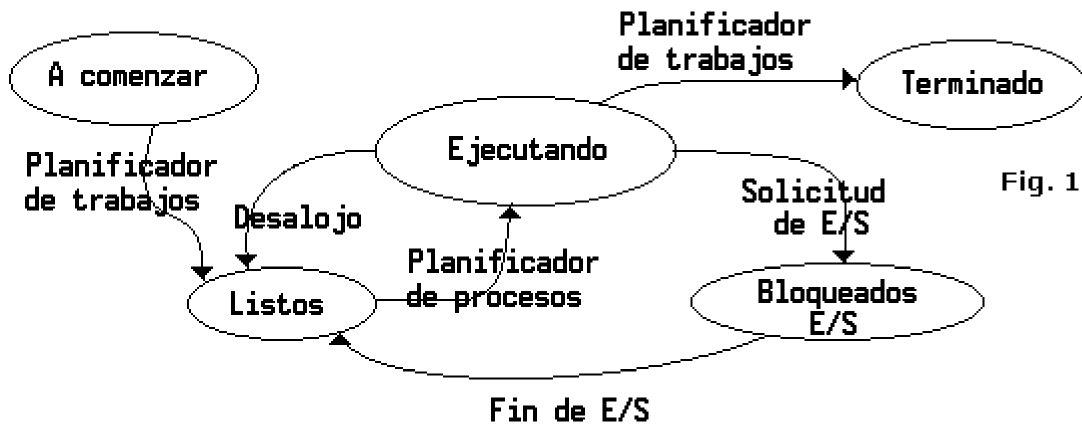


Fig. 12.12.

Sus funciones principales son :

- Seleccionar trabajos a ingresar al Sistema.
- Asignar recursos (solicitándolos a los administradores correspondientes)
- Liberar recursos (ídem anterior)

Si posee datos suficientes, el Planificador de Trabajos, puede planificar la carga de un sistema.

Esta última capacidad carece de sentido si se está trabajando en un sistema que pierde su característica de "batch".

La administración de la cola de listos que es en donde se centrará nuestro estudio, es también llamada muchas veces planificador de bajo nivel, y se lo suele denominar **Planificador del Procesador o Planificador de Procesos**.

El Planificador de procesos es el que tiene que inspeccionar la cola de listos y seleccionar, de acuerdo a algún criterio, cual de los procesos que se encuentran allí hará uso del procesador.

Para administrar eficientemente esta situación se cuenta con un conjunto de módulos, entre los cuales el primero que se visualiza es el **Planificador del Procesador**, que es el que aplica la política de selección, luego sigue un **Controlador de Tráfico**, que es el que realiza el manejo de las tablas, y además (dependiendo de la bibliografía que se consulte) un **Dispatcher**, que sería el que pone en estado de ejecución al programa, o sea carga en el procesador su contexto.

Dado un proceso que pide un servicio sobre un recurso (por ejemplo una E/S) cambiar del estado de ejecución al estado de bloqueado es una actividad que le corresponde al Controlador de tráfico, que es el que maneja las tablas.

El seleccionar el próximo proceso a ejecutar le corresponde al que aplica la política de selección, que es el Planificador de Procesos.

Y el que realmente realiza la operación final de cargar los registros, la palabra de control y los relojes que sean necesarios (dependiendo de la política de administración del procesador o sea el contexto), es función del Dispatcher.

Es muy importante tener bien en claro cuáles son las funciones :

- Manejo de las tablas,
- Selección del proceso de la cola de listos, y
- Poner en ejecución al proceso.

12.12 - **Políticas de asignación**

Los criterios para seleccionar un algoritmo de asignación del procesador deben responder lo mejor posible a las siguientes pautas :

- <u>Utilización del procesador</u>: 100 %

- Troughput: Cantidad de trabajos por unidad de tiempo
- Tiempo de Ejecución: Tiempo desde que ingresa un proceso hasta que termina
- Tiempo de Espera: Permanencia en Listos
- Tiempo de Respuesta: Tiempo que tarda en obtenerse un resultado

Luego, el mejor algoritmo será el que maximice las dos primeras pautas y minimice las 3 siguientes.

12.12.1 - **FIFO o FCFS**

Entre las políticas que puede aplicar el planificador de procesos para la selección de procesos que deben pasar del estado de listos al estado de ejecución existe obviamente la más trivial, como siempre, que es la FIFO (first-in first-out) o FCFS (first come first served). Que significa que el primero que está en la cola es el primero que va a usar el recurso procesador.

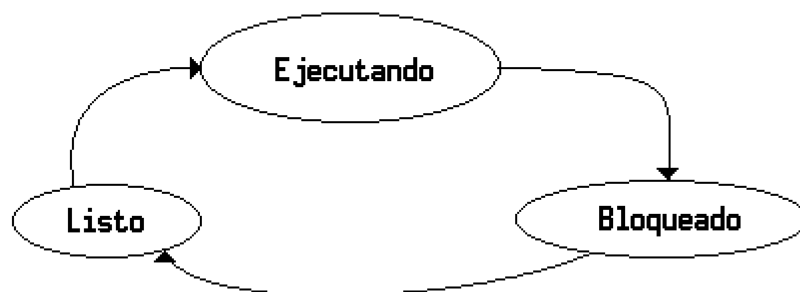


Fig. 12.13. - Administración del procesador FIFO.

Su esquema es el que se puede ver en la Fig. 12.13.

De un estado listo, pasa a un estado de ejecución, y de ese estado de ejecución el proceso abandona el recurso procesador solo por decisión propia pasando al estado de bloqueado, y después, una vez satisfecha su necesidad pasa otra vez al estado de listo. O sea que no es desalojado del uso del recurso procesador ya que una vez que lo toma lo sigue usando.

En el momento en que se produce una interrupción por fin de E/S, se atenderá ese fin de E/S el que momentáneamente hará que el proceso abandone el uso del procesador, pero después de finalizada la atención de tal interrupción, el proceso original retomará el uso de la CPU.

12.12.2 - **Más Corto Primero (JSF) Sin Desalojo**.

Una de las políticas que siempre da el mejor resultado para aquello que se quiere ordenar en función del tiempo es la del más corto primero (Job Short First).

Esto implica ordenar los distintos procesos de acuerdo al tiempo que van a necesitar del recurso procesador. O sea, la cola se ordena en función de las ráfagas que se espera que van a emplear de procesador lo distintos procesos (Nota: Ráfaga o Quantum es el tiempo continuo de uso del procesador por parte de un proceso que va desde que éste toma el procesador hasta que lo abandona por algún evento).

Este algoritmo es perfecto, ya que si se hace cualquier medición, por ejemplo del turnaround, da siempre mejor que cualquier otro algoritmo de administración.

La dificultad radica en que es necesario conocer el futuro. En la práctica es casi imposible saber con anterioridad cuánto tiempo de procesador va a necesitar un proceso. Lo que se puede hacer es calcular lo que se presume que va a utilizar.

La forma de implementarlo sería, conociendo esa medida, calificar al proceso, es decir si se tiene :

Proceso	Tiempo
P1	5
P2	3
P3	4

la cola de listos ordenada quedaría como P2, P3 y P1.

No se tiene en cuenta el tiempo total de ejecución, sino el tiempo de la ráfaga. O sea cada uno de los segmentos de uso del procesador.

Puede haber varios criterios para conocer la ráfaga. Uno de ellos sería que alguien declare un determinado valor.

Otro criterio sería, medir la cantidad de tiempo en la cual el proceso 1 ejecuta una ráfaga, supóngase que son 5 unidades de tiempo, y calificarlo luego como 5. En realidad, como no se puede conocer el futuro, lo que se hace es llevar algún tipo de estadística sobre lo que pasó en pasos anteriores. Cuando el proceso ingresa por primera vez es indistinto que se lo coloque primero o último, aunque convendrá ponerlo primero para que comience a conocerse su historia.

Hay distintas formas de calcular la calificación, se pueden llegar a tener promedios de calificaciones anteriores, por ejemplo :

$$T_{n+1} = \S\ T_n + \ldots + (1-\S)\ \S\ T_{n-1} + \ldots + (1-\S)^j\ T_{n-j} + \ldots..$$

en donde a § se lo designa con anterioridad. Si vale 1, significa que se tiene en cuenta la ráfaga anterior, si vale 0, la anterior no es tenida en cuenta, luego con 0 << § << 1 se pueden dar distintos pasos a los momentos históricos del proceso.

Una vez que el proceso sale de bloqueado dependerá de cuánto tiempo utilizó el procesador para que tenga otra calificación. En la política Más corto primero no importa el orden en el que entraron a la cola de listos, sino la calificación que se les ha dado.

En este algoritmo no tenemos, todavía, desalojo. Es decir que, como en FIFO o FCFS, el proceso abandona voluntariamente el uso del procesador.

12.12.3 - **Más Corto Primero Con Desalojo**.

Una variante del método del más corto primero es que exista desalojo. El diagrama de transición de estados tendría el aspecto que se puede ver en la Fig. 12.14.

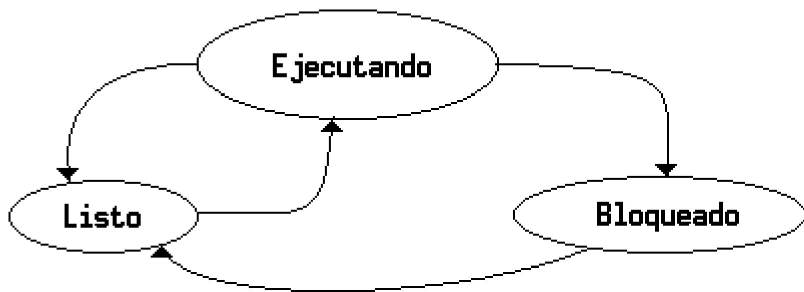


Fig. 12.14 - Más corto primero con desalojo.

Este desalojo se da a través de dos hechos fundamentales (estableciendo cada uno una política de asignación distinta):

- porque llegó un programa calificado como **más corto** que el que está en ejecución
- porque llegó un programa cuyo tiempo es menor que el **tiempo remanente**.

En el primero de los casos si se tiene en ejecución un programa calificado con 3, y en la cola de listos hay otros dos calificados con 4 y con 5, y llega en este momento, a la cola de listos, uno calificado como 2, se produce una interrupción del que está calificado como 3, se lo coloca en la cola de listos en el orden que le corresponda, y pasa a ejecutarse el programa calificado como 2.

En el caso de tiempo remanente, lo que se va a comparar es cuánto hace que está ejecutando el que está calificado como 3. Es decir, dada la misma situación del ejemplo anterior, si llega uno calificado como 2, pero resulta que al que está ejecutando sólo le falta una unidad de tiempo de ejecución, no se lo interrumpe, y el nuevo programa pasa a la cola de listos en el orden que le corresponda.

Si al programa que está ejecutándose le faltarán más de 2 unidades de tiempo de ejecución, se interrumpe, se lo pasa a la cola de listos, y pasa a ejecutarse el nuevo programa.

12.12.4 - **Administración por Prioridades**

Otro tipo de administración del procesador consiste en dar prioridades a los procesos. De alguna manera, el algoritmo del más corto primero con desalojo constituye uno de estos.

La idea es que algunos procesos tengan mayor prioridad que otros para el uso del procesador por razones de criterio.

Estos criterios pueden ser de índole administrativa o por el manejo de otros recursos del sistema. Ejemplos concretos serían :

- Por prioridad administrativa por ejemplo el "Proceso de Sueldos"
- Por recursos : Administración de Memoria Particionada; por ejemplo los procesos se ejecutan en particiones de direcciones más bajas tienen mayor (o menor) prioridad que los procesos que se ejecutan en las particiones de direcciones más altas

En estos casos es necesario implementar un mecanismo que evite un bloqueo indefinido para aquellos procesos que tienen las más bajas prioridades.

Una solución puede ser que a medida que transcurre el tiempo la prioridad de los procesos relegados se incrementa paulatinamente.

12.12.5 - **Round-Robin**

Otra forma de administrar el procesador es el Round-Robin. El origen del término Round-Robin proviene de que antiguamente en los barcos de la marina cuando los oficiales deseaban presentar una queja al capitán redactaban un documento el pie del cual estampaban sus firmas en forma circular, de forma tal que era imposible identificar cual de ellos había sido el promotor de tal queja.

Esta administración consiste en dar a cada proceso la misma cantidad o cuota de uso del procesador. Se puede visualizar esto como una calesita de la forma de la Fig. 12.15.

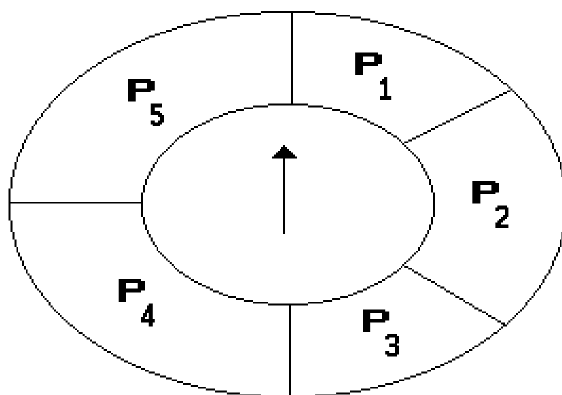


Fig. 12.15. - Calesita circular del método Round-Robin.

La asignación se comporta como una manecilla que recorre el segmento circular (que representa la ráfaga asignada) y que al pasar al próximo proceso genera una interrupción por reloj. Si el próximo proceso no se encuentra en estado de listo, se pasa al siguiente y así sucesivamente.

Es decir, a todos los procesos se les da un quantum, que es una medida del tiempo que podrán usar el procesador antes de ser interrumpidos por reloj. Obviamente puede ocurrir que cuando le toque al siguiente proceso, éste se encuentre bloqueado por operaciones de E/S, en tal caso se pasa al siguiente proceso, y el anterior tendrá que esperar que le toque el procesador nuevamente la próxima vuelta.

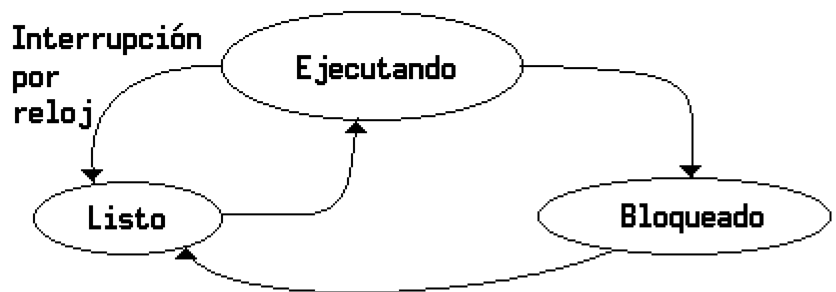


Fig. 12.16. - Administración del procesador Round-Robin.

12.12.6 - **Multicolas**

Otra forma de prioridad es asignar distintas colas y distintos quantums dependiendo del tipo de bloqueo al que llega el proceso (lectora/impresora antes que cinta) o por las características del proceso. Veamos el ejemplo de la Fig. 12.17.

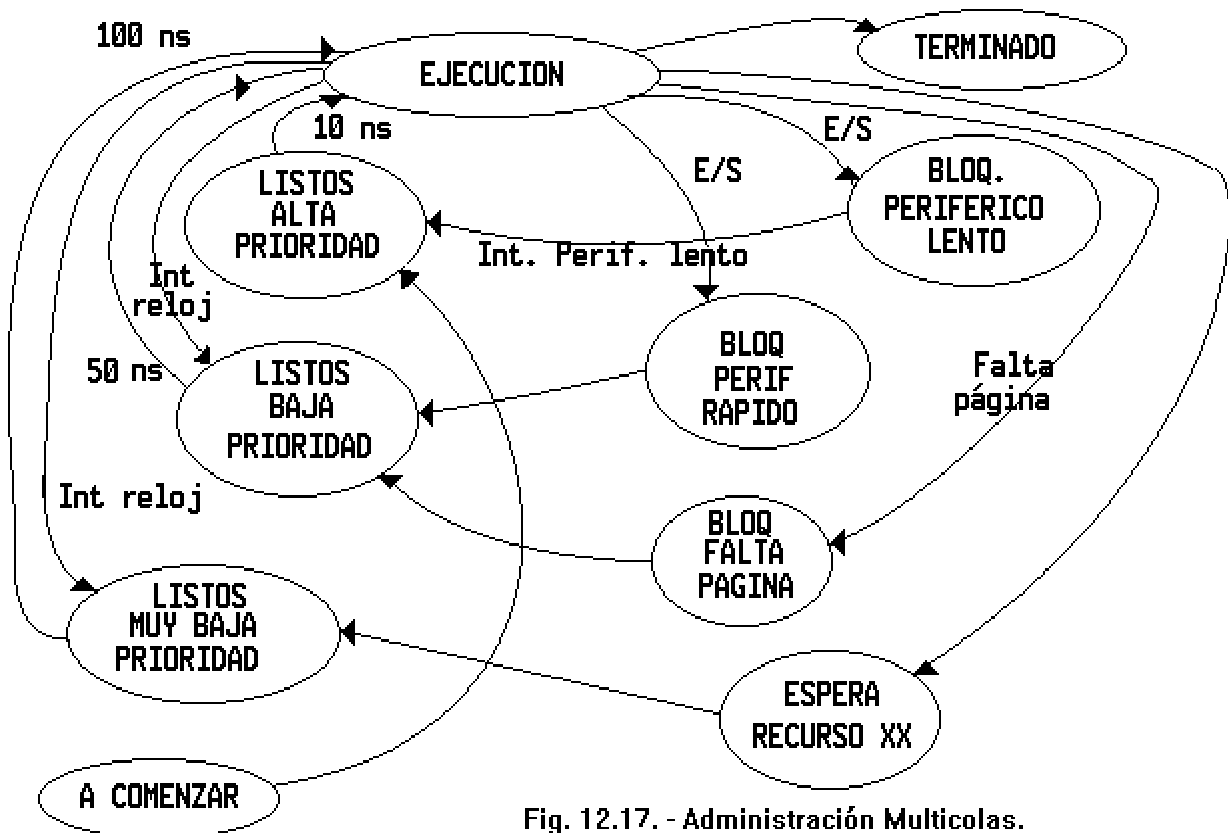


Fig. 12.17. - Administración Multicolas.

A este esquema se le podría agregar que el programa en ejecución cuando es interrumpido vaya a alguna de las colas perdiendo el uso del procesador. De esta forma se está beneficiando a los de alta E/S, en caso contrario se guardaría el valor del reloj de intervalos.

Además se podría considerar que cada uno de las colas es administrada por medio de una política diferente.

Las variantes del Round-Robin y Multicolas pueden ser :

- Todos los procesos tienen el mismo Quantum
- Si un proceso usó poco procesador se lo puede colocar por la mitad de la cola
- Si un proceso acaba de ingresar se le otorga más tiempo
- Si un proceso ejecutó muchas veces hasta el límite más alto de quantum sólo se le permitirá ejecutar cuando no haya otro proceso
- Dar preferencia a los procesos que realizan mucha E/S
- Dar preferencia a los procesos interactivos
- Dar preferencia a los procesos más cortos
- Pasar procesos de una cola a otra según el comportamiento que demuestren
- Se puede seleccionar de las colas de acuerdo a su historial

De todas maneras en los sistemas Multicolas no debe olvidarse que existe la posibilidad de procesos que queden relegados, luego es necesario implementar alguna variante para solucionar esto. Más adelante se discute una de ellas.

Este tipo de administración logra lo que se denomina un *"Balance General del Sistema"* debido a que compensa la utilización de los recursos respecto de la ejecución de los procesos otorgando mayor prioridad de ejecución (cola de Listos de mayor prioridad) a los procesos que utilizan los periféricos más lentos y viceversa.

Nótese en el grafo que los procesos que utilizan un periférico lento luego de finalizar su E/S retornan a la cola de Listos de mayor prioridad. Esto permite que dicho proceso luego de estar durante bastante tiempo fuera del circuito de listo-ejecutando (su E/S demora más ya que el periférico es más lento) pueda tener pronto una chance de retomar el uso del recurso Procesador. Si se lo hubiera colocado en una cola de listos de menor prioridad debería esperar que las colas anteriores se vaciaran antes de poder acceder al procesador.

Las colas de menor prioridad que contienen a los procesos que realizan E/S sobre periféricos veloces se encuentran cargadas continuamente ya que el proceso demora muy poco tiempo en realizar su E/S (ya que el periférico es rápido) y vuelve rápidamente a la cola de Listos, en tanto que las colas de mayor prioridad usualmente se vacían con bastante frecuencia ya que sus procesos demoran un tiempo considerable en volver del estado de Bloqueado.

Finalmente este mecanismo logra asimismo que los periféricos lentos se utilicen más frecuentemente logrando su máximo aprovechamiento.

12.13 - **Mediciones de performance**

Una forma analítica de comparar los distintos algoritmos es haciendo cuentas con los tiempos que se conocen de los procesos. Supongamos un ejemplo en el cual hay 5 trabajos (5 procesos) cuyas ráfagas a considerar son las siguientes :

PROCESO	RAFAGA
A	5
B	30
C	3
D	10
E	12

y propongamos evaluar cuáles son los tiempos de espera. O sea cuánto espera cada uno de esos procesos. Los guarismos son :

FIFO (un.promedio)	25.2
El más corto primero (sin desalojo)	11.8
Round-Robin (quantum = 10)	20.2

Estas cifras se calculan de la siguiente forma :

En el caso del FIFO, la primera ráfaga del trabajo A dura 5, en el caso del B dura 30, en el caso del C dura 3, etc.

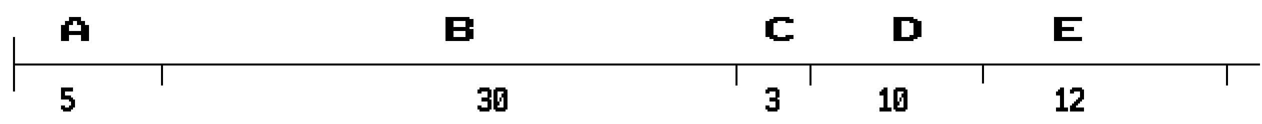


Fig. 12.18.

El primero no ha esperado nada para obtener su ráfaga, el segundo ha esperado 5, el tercero 35, etc. El cálculo final para hallar el tiempo de espera promedio sería entonces (0 + 5 + 35 + 38 + 48) / 5 = 25.2

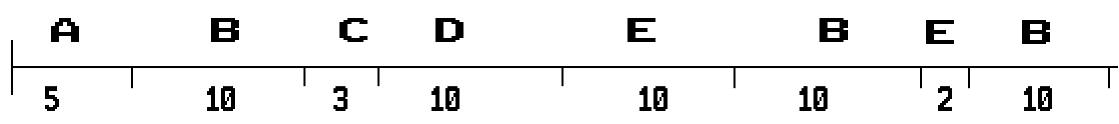


Fig. 12.19.

Para Round-Robin se tendrán pedacitos. Se tiene un quantum de 10, en el A se tiene 5, para el B se tiene 30, pero en 10 le llega la interrupción por reloj (le quedan 20 por ejecutar), para el C se tiene 3, etc.

Los tiempos de espera serán : para el A cero, para el B hay que sumar (5 + 23 + 2), para el C será 15, para el D será 18 y para el E habrá que sumar (28 + 10). El tiempo promedio de espera será igual a 20.20 (101 / 5)

El más corto primero es el mejor de todos, pero no se deben olvidar los problemas de implementación. Recuérdese que si se mezclan procesos interactivos con procesos de tipo batch de mucho tiempo de procesador estos últimos no ejecutarán nunca.

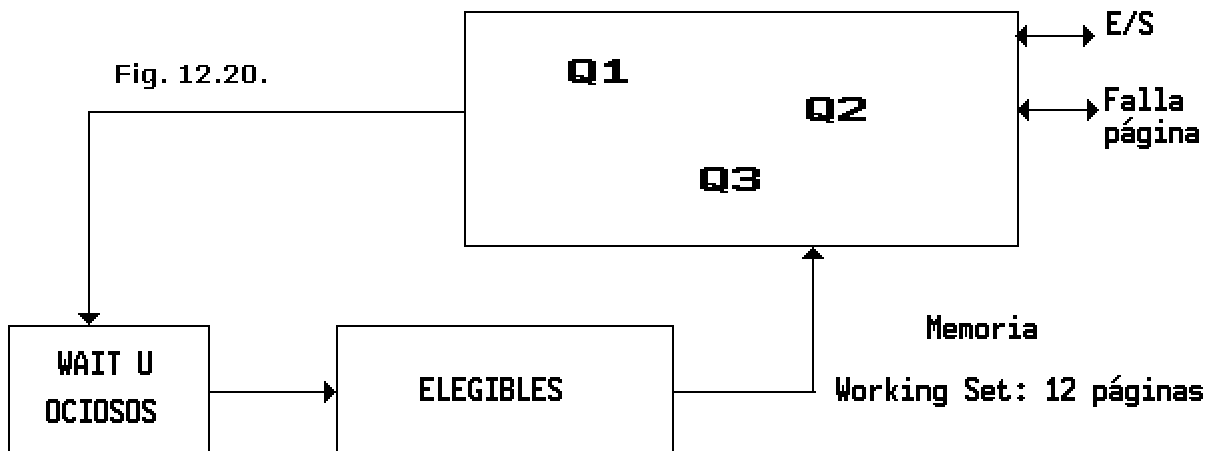
Para medir un sistema en general, lo que conviene hacer es, conociendo la carga del mismo, armar un conjunto de programas representativos de la instalación, correrlos todos juntos y obtener resultados. A partir de esos resultados, se deben tomar decisiones. Esto se conoce como **Benchmark** (banco de pruebas). Teniendo un subconjunto representativo de la carga natural de un sistema, sabiendo que el mismo tiene un 10% de programas

COBOL, 50% de programas FORTRAN, y 40% de una actividad determinada, hay que hacer un banco de pruebas con esa representación, cargar los distintos sistemas, tomar las medidas necesarias y compararlos.

Esto no está midiendo sólo la actividad del procesador, sino también cómo funciona el sistema frente a la carga que se le presenta.

12.14 - **Ejemplo: Caso Real**

Veamos un ejemplo real de una administración multicola. Estas colas se manejan por medio del algoritmo Round-Robin. Este sistema intenta premiar las actividades interactivas. Es un sistema interactivo y deja siempre con la más alta prioridad a todo aquel que tenga una interacción con su terminal. Se manejan básicamente tres colas : Q1, Q2, Q3., las cuales comparten el recurso procesador de la siguiente manera :



- dentro de Q1 la administración es Round-Robin y solamente se pasa a la Q2 cuando Q1 está vacía o cuando todos los procesos que están dentro de la Q1 están bloqueados. Lo mismo ocurre con la Q2 y la Q3.

En el Round-Robin no importa si un proceso está bloqueado. La representación en forma de calesita es la de la Fig. 12.21.

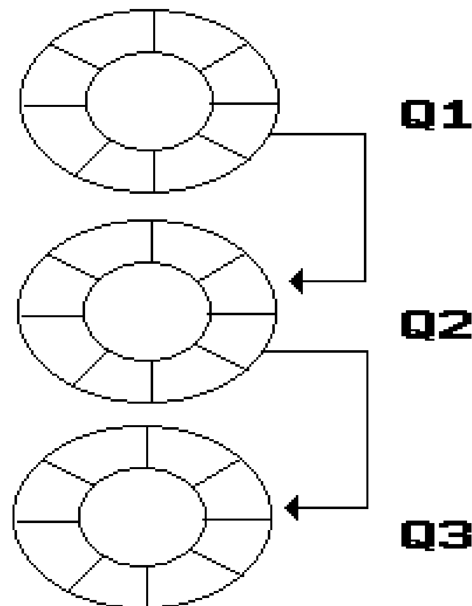


Fig. 12.21.

El proceso 1 es el que toma el recurso procesador, ejecuta lo que necesita y después se pasa al proceso 2. Si éste está en estado de bloqueado, automáticamente, se pasará al proceso 3, o sea se sigue estando dentro de la lista. En el caso de Multicolas ocurre lo mismo, dentro de la Q1 va a haber procesos que van a estar en estado de bloqueado, por razones de E/S. La calificación de cada una de estas colas sería que los procesos que están en Q1 son altamente interactivos (tienen mucha actividad con la terminal) y los procesos que están en Q2 tienen un poco menos. Los de Q3 tienen poca actividad con la terminal (por ejemplo una compilación estaría en Q3).

Existe la posibilidad de salir de estas colas y entrar a la cola de Elegibles. Allí se encuentran procesos a los que les falta el recurso procesador, han excedido en mucho el uso del recurso procesador, o están ociosos. Antes de poder volver a pelearse con el resto de los procesos por el uso del procesador, tienen que pasar una especie de "prueba de fuego" que está relacionada con la memoria.

Existe una medida de memoria llamada **Working Set**, que para este sistema en particular está midiendo el número de páginas esperadas en memoria cuando éste proceso toma el recurso procesador para que trabaje. Sería la cantidad de páginas promedio que necesita para trabajar, cada vez que este proceso toma el control del procesador.

Para poder volver a competir para ocupar el procesador, cuando un proceso está calificado con un Working Set de **n**, no lo puede volver a hacer, mientras la cantidad de bloques de memoria real no sea mayor o igual que esta cantidad (n). Si un proceso está en elegibles porque usaba mucho procesador y está calificado con un working set de 12, no va a poder ingresar a ninguna de las colas a menos que la cantidad de páginas no sea menor o igual que la cantidad de bloques libres que hay en memoria.

Aquí se lo empieza a penalizar, no solo por haber usado mucho el recurso procesador, sino por haber estado utilizando demasiada cantidad de páginas.

El paso de una cola a otra es la siguiente :

	Quantum	Sale de cola Qi	Entra a cola Qi
Q1 Interactivo	8	Int. por reloj	Int. Terminal
Q2 No-Interactivo	64	Int .por reloj (*6)	sale de Q1
Q3 Pesados	512	Int.por reloj(*8)	sale de Q2

La cola 1 es interactiva (por definición). El quantum que se le da es de 8 unidades de tiempo y sale de la cola 1 cuando excedió ese quantum, o sea por interrupción de reloj. Pero lo importante es que se vuelve a entrar a la cola 1 por interrupción de terminal. No importa dónde se encuentre, va a volver siempre a la cola uno cada vez que exista una operación de E/S desde la terminal. O sea se están priorizando las operaciones por terminal.

La cola 2 es menos interactiva que Q1; el quantum es de 64 unidades de tiempo (generalmente se colocan como primer quantum un número cualquiera y luego múltiplos de éste porque dependen de cada modelo de máquina), y sale por interrupción de reloj, pero por 6, o sea seis veces ha sido interrumpido mientras estaba en la cola 2 por reloj. En Q1 cuando llega una sola interrupción por el quantum que teníamos, pasa a Q2. Pero en Q2 se le da bastante más tiempo y además soporta hasta 6 veces estas interrupciones, o sea permanece más tiempo en esa cola menos interactiva.

La Q3 tiene los procesos más pesados. Su quantum es de 512 unidades, interrupción por reloj, pero 8 veces, y entra porque sale de Q2. Entonces en el momento que excedió las 8 interrupciones de ráfagas de 512 cada una, es cuando tenemos la salida de cola y se pasa a la cola de Elegibles.

Si Q1 y Q2 están muy llenas, o sea están con muchos procesos, posiblemente los demás procesos que han caído a la cola 3 no entren "más" al recurso procesador. Es el problema del efecto pila, han quedado abajo y no toman nunca más el recurso procesador.

12.15 - **Efecto Residual**

El problema de los procesos que están en el resto de las colas, es que hay que encontrar un mecanismo para que vuelvan a tener derecho a tomar el recurso procesador, para evitar el efecto residual de los procesos que no lo tomarían nunca.

Para evitar ese defecto lo que se hace es en el momento en que el proceso abandona el recurso procesador porque fue interrumpido o por que el mismo decide abandonarlo, se realiza una cuenta de cuándo debería ser la próxima vez que debería tomar el recurso procesador. Cada cierta cantidad de tiempo se controla si existen procesos que se encuentran con una hora de próximo proceso muy atrasada. El cálculo, podría ser considerando el ejemplo anterior, si estuviera nada más que en Q1 :

$$\text{Hora próxima procesador} = \text{Hora actual} + (n-1)_{Q1} * \text{Quantum}_{Q1}$$

siendo $(n-1)_{Q1}$ los (n-1) procesos de la cola 1.

Si estuviera en la cola 2:

$$\text{Hora próxima procesador} = \text{Hora actual} + (n)_{Q1} * \text{Quantum}_{Q1} + (n-1)_{Q2} * \text{Quantum}_{Q2}$$

Si el proceso uno abandona el recurso procesador en un determinado instante, se toma la hora actual, se mira qué cantidad de procesos hay en esa cola y se lo multiplica por el quantum, o sea se lo multiplica por cuanto debería esperar una vuelta completa.

Puede suceder que existan procesos en Q2 y Q3 cuya hora de próximo uso de procesador ya haya pasado hace un rato, esos son procesos que quedaron en espera. Entonces para evitar el problema pila, el efecto residual, lo que se hace es examinar las colas y, cuando se detectan procesos que se encuentran en esta situación, se les asigna el procesador por más que haya procesos a los cuales les correspondan su uso en colas de más alta prioridad.

De la misma manera que se examinan las colas 1, 2 y 3, para detectar el problema de los procesos que quedan en forma residual, se hace exactamente lo mismo con los procesos que están en la cola de elegibles. Aquí el impedimento es su working set, y se usa una pequeña trampa que es descontar a ese working set del proceso un determinado porcentaje (puede ser 10%). Cada vez que se lo examine y se determine que hace rato que está ahí, la próxima vez se le descontará un 10%, hasta que realmente esa cifra pueda llegar prácticamente a cero y pueda tomar el recurso procesador.

Cuando se pasa un proceso a la cola de elegibles no se calcula más la hora próxima de procesador, pues de allí se extraen con el cálculo del Working Set.

Para que este sistema funcione bien, toda la actividad del SO. (incluida la paginación, las administraciones de otras colas, las operaciones de E/S, etc.) no debe exceder su permanencia en el procesador más del 5% de su tiempo total. Si se excede comienzan los problemas.

12.16 - **SEMAFOROS**

Evidentemente, si existe más de un proceso ejecutando en una arquitectura SIMD o MIMD que de distintas maneras pueden llegar a acceder al mismo lugar de memoria se genera un problema de administración de la concurrencia. Entendiendo como concurrencia el intento de acceso simultáneo a una misma variable, un mismo dato, una misma dirección de memoria.

Para evitar el problema de que se cambie la información de una manera no deseada, lo que se debe hacer es establecer un sistema de protección para posibilitar que esta concurrencia se dé, pero en forma ordenada. O sea que accedan en orden, de a uno por vez.

La forma de realizar esto, es la de colocar algún tipo de señal, y que esa señal sea respetada, de tal manera que si dice que un proceso no puede acceder a un determinado lugar de memoria, no lo haga.

Pero además esa señal debe comportarse de tal manera que una vez que un proceso ha hecho uso de la zona restringida permita el acceso de un nuevo proceso a la misma. O sea que se busca protección y sincronización entre procesos.

Lo mejor que se puede hacer es colocar algún tipo de bandera, de flag; algo que se denomina, normalmente, semáforo.

Definición : Semáforo es un conjunto de bits que va a indicar un determinado estado que indica si se puede o no acceder a un determinado dato, o a una determinada posición de memoria, o si se puede realizar algún tipo de acción.

Habría dos acciones para manejarse con un semáforo. Una sería cerrar el semáforo porque se va a acceder a ese lugar, y una vez que se accede, liberarlo, o sea ponerlo en verde para que pueda acceder algún otro proceso.

La forma de hacerlo la llamaremos cierre y escribiremos

CIERRE (X)

para indicar que se cierra el semáforo X. Obviamente, si lo que se quiere hacer es controlar que no se está intentando acceder a un lugar que no corresponde, en ese momento lo que habrá que hacer es mirar ese valor de X, y lo notaremos :

EXAMINAR(X)

Establecemos la siguiente convención : si X es igual a 0, es de libre acceso; y si X es distinto de 0, significa que está ocupado.

Entonces se examina X y si X es igual a 0, se toma y se realiza la acción; si X es distinto de 0, se debe esperar. La liberación, APERTURA(X), sería, sencillamente, asignarle a X una valor igual a 0.

Sea el siguiente programa pequeño de la rutina CIERRE(X):

```
1:  If X not = 0   Then  LOOP(X)
                   2:  X <<--- 1
                   3:  Tomar el recurso
```

Examinar X será preguntar por el valor de X, o sea preguntar si X es igual a 0. Si resulta afirmativamente, lo que se hace es asignarle a X un valor cualquiera distinto de 0, o sea, ocuparlo y luego vendrá la acción de tomar el recurso. Y si X no vale 0, la acción a tomar será esperar, es decir examinar nuevamente el valor de X hasta que cambie.

Esto tiene un par de problemas. El primero de ellos es que, si una vez que, ante la pregunta, se sale por X = 0, y en ese momento llega una interrupción y se quedó con el estado de X igual a 0, cuando vuelva a ejecutarse ya no se pregunta nuevamente, se pone en 1 y se lo toma. Pero no se sabe lo que sucedió en el ínterin. Podría haber venido otra rutina equivalente, preguntar si X estaba en 0 (a lo que respondería que sí ya que no se llegó a actualizar previamente) le colocaría un 1 y tomaría el recurso, produciéndose luego la doble asignación. Luego, como sistema de protección no es eficiente.

El otro caso es que si se queda constantemente preguntando por el semáforo significa que se está ejecutando una instrucción, o sea se está utilizando procesador inútilmente ya que no se está haciendo nada productivo. Y de hecho, si estuviera en un sistema de monoprocesamiento hasta que no existiera una interrupción, se quedaría eternamente preguntando por el valor del semáforo.

Atacando el primer problema, que es el asunto de la interrupción, se sabe que las instrucciones no son interrumpibles por la mitad. Entonces lo que habría que lograr es que estas dos acciones, o sea la pregunta y la asignación del semáforo, se tradujesen en una sola instrucción. O sea generar una nueva instrucción que, simultáneamente haga el cambio del valor del semáforo, y testee el valor que tenía.

Para esto existe una instrucción que se llama **TEST-AND-SET (TS**).

Nótese aquí que lo que se hace es cambiar el hardware, que haga la observación de cuánto es el valor del semáforo, y además asigne el valor de ocupación.

Esta instrucción TEST-AND-SET se podría traducir de la siguiente manera :

```
CRB1   5,X              (Cargar registro base número 5 con valor X)
COMPR 5,0               (Comparar el contenido del registro 5 con valor 0)
```

El funcionamiento sería cargar en el registro 5 el valor que tiene el semáforo X y ponerle en ese lugar un 1.

Con esta instrucción, ya no importa lo que suceda después, pues si luego llegara una interrupción, una de las primeras cosas que se harían sería salvar los registros, y por ende el registro 5 en particular, y además, ya también se cerró el semáforo sin todavía saber cuál era el valor del mismo.

Luego, lo que se hará será comparar el registro 5 contra el valor 0. Si el semáforo valía 0, en el momento que se ejecutó la primera instrucción, ya vale 1; y se entera cuál era el valor anterior del semáforo en el momento que testee el registro 5.

Si el semáforo valía 1, o sea que ya estaba cerrado, se le volvió a escribir un 1, o sea que no cambió su estado. Y tampoco importa qué es lo que va a suceder después porque, en realidad, se va a testear ese 1 en el registro 5. O sea que de esta manera se esta asegurando que las interrupciones no afectarán el resultado de ese fragmento de código.

El otro problema era que se quedaba ciclando, preguntando por el semáforo, y se consumía mucho tiempo ejecutando esa instrucción que, hasta que no venga una interrupción no se podía abandonar (si estamos en mono-procesamiento).

Para eso, lo que se aplica es lo siguiente : dada esa situación, poder pasar a un estado de espera; donde ese estado de espera implica que se encuentran esperando la apertura de ese semáforo. Se puede usar una instrucción WAIT(X), que lo que haría sería que poner en una cola asociada al semáforo X a la rutina que se quedó esperando la apertura de ese semáforo. Cola que tiene una estructura muy sencilla.

X ------------>> Procesos en espera de apertura de X
....
....

En el momento en que se hace la apertura de ese semáforo, habría que examinar esta cola para saber si realmente hay procesos que estén esperando la apertura de ese semáforo. Para eso existe otra primitiva que se denomina SIGNAL(X), que toma al primer proceso que se encuentra a la espera de ese semáforo y lo manda a la cola de listos. O sea que hace la función de despertar procesos.

Entonces, las rutinas de cierre y de apertura quedarán de la siguiente forma :

	CIERRE(X)	APERTURA(X)
EXAM:	T.S.(X)	X = 0
	If X = Ocupado then WAIT(X)	SIGNAL(X)
	Go to EXAM	
	Tomo recurso	

Pero, porqué no inhibir todas las interrupciones, mientras se realiza esta tarea ? El utilizar el mecanismo de inhibir todas las interrupciones en lugar de utilizar la instrucción Test-and-Set tiene sus desventajas. Supongamos que se están sincronizando procesos, básicamente esos procesos van a tener una zona crítica, que es donde no se desea que ocurran problemas de concurrencia, y seguramente, después, otra zona que deja de ser crítica nuevamente. La extensión de esto pueden ser muchísimas cosas, por ejemplo, como retirar información de buffers , etc. Luego, si se inhiben todo tipo de interrupción, se ejecuta sin problemas, pero se estarían perdiendo eventos que están sucediendo.

Existe un caso en el cual tiene bastante sentido que el procesador se quede ciclando sobre un semáforo, y no que se produzca un Wait. Este es el caso en el cual se tiene, por ejemplo, dos procesadores que quieren acceder a la cola de listos para seleccionar el próximo proceso a ejecutar. En un momento dado está uno de ellos seleccionando por tanto se está ejecutando el planificador de procesos, y en ese mismo instante, también en el otro procesador se quiere ejecutar el planificador de procesos.

Uno de los procesadores justamente tiene que ir a la cola de listos para seleccionar el proceso que tiene que ejecutar. Pero el otro está haciendo la selección y tiene el recurso (cola de listos) tomado. Si no puede elegir, no tiene sentido que se espere nada, porque lo que quiere hacer es justamente tomar un proceso para continuar la ejecución. Este sería un caso en el cual uno de los procesadores se quedará loopeando en la pregunta esta de ver cuándo cambia el valor del semáforo. Esos semáforos reciben el nombre de SPIN.

12.16.1 - **Semáforos Contadores**

Dijkstra ideó lo que se denomina semáforos contadores, con el siguiente concepto: los semáforos pueden servir también para contar la cantidad de procesos que hay en espera de un recurso. Además con la idea de ir después contando algunas otras cosas.

Para ello, inventó dos operadores, para trabajar sobre los semáforos. Uno de ellos es el operador P, cuya función es la siguiente :

$x = x - 1$

si $x < 0$, se espera.

Esto considerando un valor inicial de $x = 1$. El otro operador es el operador V, que correspondería a la apertura del semáforo, que hace :

$x = x + 1$

si $x \leq 0$, despierta.

Por supuesto que la espera lo podemos transformar en un WAIT (X),y el despierta en un SIGNAL(X).

P(X)	V(X)	Valor inicial $x = 1$
$x = x - 1$	$x = x + 1$	
If $x < 0$ WAIT(X)	If $x \leq 0$ SIGNAL(x)	

Entonces, si se tuviera un buffer, sobre el cual alguien va a poner información, que llamaremos emisor, y algún otro que llamaremos receptor, retira información, se puede controlar, si el lugar en el cual se desea colocar información está libre y además, se puede controlar si existe la información que se desea. O sea se controla y se sincroniza la actividad de ambos procesos (Caso del Productor-Consumidor).

La idea es la siguiente : no se quiere que el emisor escriba antes que el receptor haya retirado la información, y tampoco que el receptor retire información que el emisor todavía no escribió.

Fig. 12.19. - Caso del Productor-Consumidor

La rutina que emite, podría tener un semáforo que controlara el espacio. Una vez que colocó la información, la cierra, es decir no se puede colocar más información ahí, y abrir un semáforo que habilitaría la extracción de información. Con lo cual podría realizar lo siguiente:

	EMISOR	RECEPTOR
Mensaje = 0	P(Espacio)	P(Mensaje)
Espacio = 1	.	.
	Coloca inf. en el Buffer	Retira
	.	.
	V(Mensaje)	V(Espacio)

El receptor, tendría que preguntar si ya existe el mensaje. En el caso en que existiera, retirarlo y volver a habilitar el espacio.

Si se piensa un ejemplo en el que se deposite información en un vector (Vec(i)) de dimensión 3 se puede tener un par de variables, i=0, j=0, que van diciendo a cada uno de los módulos, el emisor y el receptor, poner la información que es remitida. La idea es no tapar información que no ha sido retirada, no retirar información que no existe y llegar a la situación de acceder en forma exclusiva a esta zona de memoria.

La rutina de emisión es la siguiente :

E — Mensaje = 0
P(Espacio) — Espacio = 3
VEC(i) = M
i = (i+1) módulo 3
V(Mensaje)

en VEC(i) se está poniendo el mensaje del emisor, después se incrementa i con i+1 pero con módulo 3 para no superar la cantidad total de elementos que contiene el vector. Y que después se habilita la posibilidad de retirar un mensaje.

El receptor hace la operación contraria :

R
P(Mensaje)
X = VEC(j)
j = (j+1) módulo 3
V(Espacio)

toma de alguna manera la información desde VEC(j), luego lo incrementa en uno (todo esto también módulo 3) y avisa de un espacio nuevo para seguir colocando información.

Y esto funciona muy bien mientras no exista la posibilidad de que ocurra una interrupción y nuevamente antes de poder actualizar el subíndice, con el cual se está recorriendo el vector, una nueva tarea quisiera utilizar este procedimiento de emisión.

Luego lo que se hace para solucionar esto es colocarle un semáforo exclusivo (Exclu), cuyo valor inicial es 1, y después se lo habilita. Con el emisor se hace lo mismo.

E
P(Exclu)
P(Espacio)
VEC(i) = M
i = (i+1) módulo 3
V(Mensaje)
V(Exclu)
R
P(Exclu)
P(Mensaje)
X = VEC(j)
j = (j+1) módulo 3
V(Espacio)
V(Exclu)

Hasta aquí todo va bien, pero si los valores fuesen :

Espacio = 0 Mensaje = 3 Exclu = 1

significa que hay información en todo el buffer y no se puede seguir colocando porque sino se taparía la que existe, el receptor no está retirando información, y por ende se produce un problema.

Si E hace un P(Exclu), o sea Exclu que en ese momento permite la entrada (vale 1), pero luego de realizar P(Exclu) , éste queda con valor 0 y cuando hace P(Espacio), se da cuenta de que no puede entrar (está en -1). Si R hace P(Exclu) estando E interrumpido a la altura que corresponde entre P(Espacio) y VEC(i) = M y el receptor intentando retirar información, hace que con su P(Exclu), Exclu tome valor (-1) y se interrumpe ahí, nos encontramos ante la situación de que el emisor (E) está esperando que alguien retire información mientras el receptor está esperando tener la exclusividad de acceso para poder retirar la información

Lo que hemos armado es un **abrazo mortal**, y allí quedó trabado el sistema de sincronización.

El verdadero problema de esto es que en realidad se está exigiendo una condición, la de exclusividad, previa a la de conocer si realmente corresponde entrar a la zona exclusiva, o sea se está pidiendo exclusividad de algo cuando aún no sé sabe si puede utilizarse.

Con el pequeño cambio de preguntar primero si corresponde entrar y después preguntar si se tiene el recurso en forma exclusiva, se ha arreglado el problema, o sea lo que se hace es :

E
P(Espacio)
P(Exclu)
VEC(i) = M
i = (i+1) módulo 3
V(Exclu)
V(Mensaje)

y para el receptor se hace lo mismo, primero se pregunta si puede entrar, y después pregunta si lo tiene exclusivo, y cuando se va hace exactamente lo mismo

R
P(Mensaje)
P(Exclu)
X = VEC(j)
j = (j+1) módulo 3
V(Exclu)
V(Espacio)

Y ahora no existe el problema anterior porque lo primero que se hace es preguntar si realmente corresponde entrar, y una vez que se puede, se pide con exclusividad el uso del recurso. Entonces al implementar el orden correcto, alcanza con un solo semáforo.

El Exclu sirve para dejar una zona de exclusividad y para escribir o retirar algo. La razón de entrar en exclusividad es porque si un proceso se interrumpe en el medio y a continuación ingresa otro que realiza una actividad del mismo estilo podría destruir información que tenía el primer proceso.

Este tipo de problema es la razón por la cual los semáforos en algunos textos son bastante criticados, y se proponen algunos otros tipos de solución.

Una solución son los monitores que es como tener una especie de programas de servicios que se encarguen de realizar la tarea de exclusividad y sincronización.

12.16.2 - **Productor-Consumidor (Implementado con Stack)**

Veamos otro ejemplo de Productor-Consumidor, pero implementado con stack, donde se puede apreciar mucho mejor la necesidad del semáforo de exclusión.

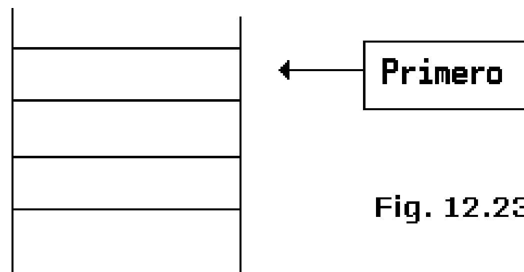
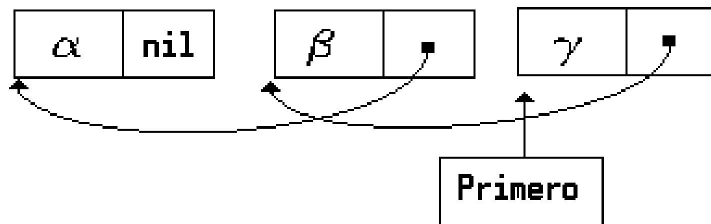


Fig. 12.23.



Supóngase que se desea una cantidad máxima de mensajes (que aquí llamaremos MAX) y que se cuenta para ello con una estructura encadenada (ver Fig. 12.23) y con la siguiente información:

Primero (apuntador) = Nil (apunta al último mensaje generado o primero a consumir),
P(apuntador) = dirección del mensaje que se está produciendo,
C (apuntador) = dirección del mensaje que se está consumiendo,
E (semáforo) = MAX = 3 (cuenta el máximo de mensajes producibles)
S (semáforo) = 0 (cuenta el número de mensajes disponibles para consumo)
X (semáforo de exclusión) = 1

Las rutinas serán :

	A (Productor)		B (Consumidor)
	P(E)		P(S)
	P(X)		P(X)
[1]	P = genera apuntador	[2]	C = Primero
	P.Mensaje = Dato		Primero = C Próximo
	P. Próx = Primero		Dato = C.Mensaje
[3]	Primero = P		Libera C
	V(X)		V(X)
	V(S)		V(E)

Ambas rutinas cuentan con el semáforo de exclusión X.

Supóngase momentáneamente que no se tiene el semáforo X, y que los valores de las variables luego de emitidos dos mensajes son los indicados en la Fig. 12.24 situación 1).

La rutina A comienza a producir el tercer mensaje y llega a ejecutar hasta la instrucción [1] luego de lo cual es interrumpida. Poco tiempo después comienza a ejecutarse la rutina B la cual llega a ejecutar hasta la instrucción [2] inclusive antes de ser interrumpida también.

Retoma la rutina A que ejecuta hasta la instrucción [3] inclusive, continúa B y luego se completa A.

Nótese que se produjo el mensaje de la dirección 300 y se consumió el mensaje almacenado en la dirección 200, sin embargo el valor del apuntador al próximo

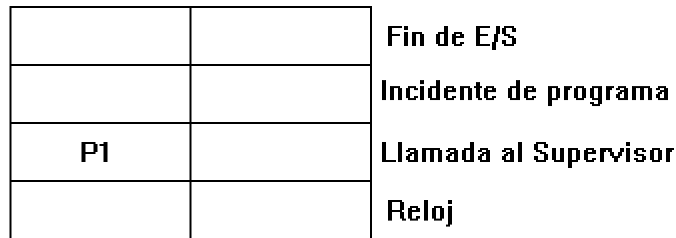
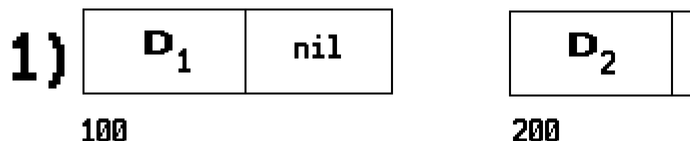
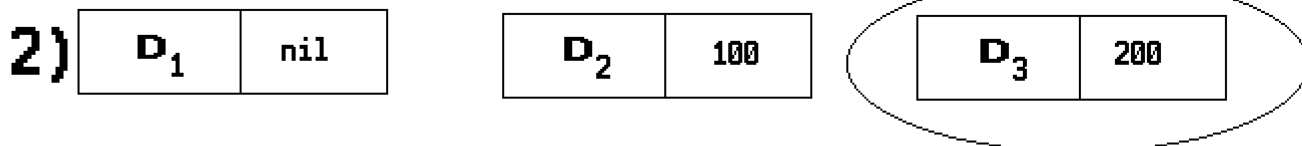


Fig. 12.25.





	Valores iniciales		1) A	B	A	B	A		Valores finales
Primero	nil	100	200			300	100		100
E	3	2	1	0			1		1
S	0	1	2		1			2	2
P	nil	100	200	300					
C	nil				200				
						2)			

Fig. 12.24.

mensaje a consumir (Primero) indica que se debe consumir el mensaje de la dirección 100. Aún cuando los contadores de mensajes indican que existían 2 mensajes para consumir se ha perdido el mensaje que se almacenó en la dirección 300.

Luego se perdieron datos a causa de la falta del semáforo de exclusión.

La implementación de semáforos es posible, y sería muy bueno que su ejecución fuera atómica, es decir que un P y un V sean directamente una instrucción. Tener implementado desde hardware una instrucción "operador P" y otra "operador V", y sencillamente cuando se les indicase sobre cuál semáforo trabajar sabrían sobre qué zona de memoria actuar.

La implementación más usual es que esto se transforme en una llamada al S.O., con lo cual se hace una interrupción, indicando que se quiere usar la rutina que se llama P y la rutina que se llama V.

Obviamente, una forma de implementarlo sería en el momento que se lo va a usar inhibir todo tipo de interrupción, eso generalmente trae problemas porque : primero se debe estar completamente seguro de que no se podrá producir un loop en lo que se hará, y segundo no siempre se pueden inhibir todo tipo de interrupción; si, por ejemplo, se tiene un sistema de control de procesos no se pueden inhibir todas las interrupciones, porque posiblemente alguna de ellas sean muy importantes.

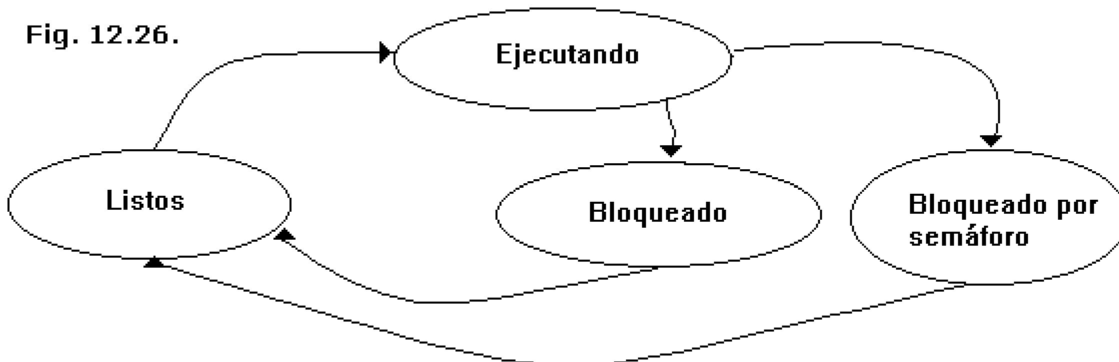
12.16.3 - **Ejemplo de administración de Procesador con semáforos**

Veremos un caso en el cual se visualizará cómo se va desarrollando el tema de ir pidiendo recursos que estén protegidos por lo menos por un semáforo, el esquema de sistema va a ser el siguiente : existe una zona donde se reciben las interrupciones

Por ejemplo el primero es el nivel de fin de E/S, otro será el nivel de incidente de programa, el siguiente el de llamada al supervisor y luego el nivel de interrupción por reloj, si es que existe, o cualquier otro tipo de interrupción que se tenga.

El núcleo de este sistema será el siguiente, todos los programas van a poder estar en los siguientes estados :

- un estado de ejecución,
- un estado de listos,
- un estado de bloqueados y
- un estado más que será bloqueado en espera de semáforo.



En este sistema supóngase un recurso único, y ese recurso está controlado por una rutina de asignación (rutina S). Esa rutina de asignación lo que hace es ejecutar un operador P sobre un determinado semáforo que llamamos X:

Rutina de asignación :
S(asignación)
P(X)
Asigna

donde tenemos que :

P(X)
X = X - 1
If X < 0   Wait (X)

Y la rutina de liberación será :

Desasignar
V(X)

donde el operador V(X) será :

V(X)
X = X + 1
If X $\leq$ 0   Signal(X)

Veamos qué sucede con este sistema en el cual en un momento determinado se tienen los siguientes estados : en ejecución, una cola de listos, bloqueados y bloqueados por semáforo, y supóngase que hay un proceso P1 que está ejecutando y en la cola de listos hay otros dos P2 y P3. La administración de la cola de listos es por lo menos FIFO, y en caso de querer una administración del procesador que también fuera FIFO, entonces se elimina la interrupción por reloj (ver Fig. 12.27).

El proceso P1 está ejecutando, y en un momento dado hace una llamada al supervisor para solicitar el recurso S, mientras tanto los procesos P2 y P3 están en la cola de listos. Al realizarse la llamada al supervisor, hay una interrupción, un intercambio de la palabra de control, la PC del programa P1 va a ir a la izquierda de la zona que tenemos asignada como llamadas al supervisor y se tomará la palabra de control que está a la derecha, que es la que corresponde a la rutina que atiende las llamadas al supervisor. En conclusión lo que se tiene aquí es la rutina de atención al supervisor, mientras que tenemos a P2 y a P3 en la cola de listos.

P1 tiene suspendida su ejecución, o sea su palabra de control está guardada. Esta rutina de atención del supervisor se tiene que dar cuenta de dos cosas:

1) Le han pedido el recurso
2) P1 que pidió el recurso S tiene que pasar a bloqueado, o va a estar esperando ese recurso.

Entonces la rutina hará dos cosas:

1) Va a preparar información sobre P1 que va a pasar a bloqueado.
2) Tiene que preparar información que genera en este momento una rutina pidiendo el recurso S para el programa 1 (Rut(S,P1), que es una nueva tarea.

El encargado en un sistema de administración de procesador de colocar los nuevos estados para un programa que se ha generado y otro que está siendo suspendido es el controlador de tráfico (CT). Entonces se va a llamar con estos datos al controlador de tráfico, todavía P2 y P3 están en la cola de listos.

El controlador de tráfico generará las dos cosas que le han pedido que haga

- poner en estado de bloqueado al P1 y
- colocar dentro de la cola de listos a la rutina que pide el recurso S para el P1.

Luego de hacer esto, el controlador de tráfico terminó su tarea, entonces se tiene que seleccionar cuál es el próximo proceso que tiene que tomar el recurso procesador. Como conclusión el planificador de procesos es el que entra a funcionar y siguen los mismos procesos en la cola de listos.

Supóngase que la cola de listos es FIFO y no da ningún tipo de prioridad a la rutina S, el PP lo que hace es seleccionar algún proceso de la cola de listos por medio de algún algoritmo, entonces selecciona a P2, y en la cola quedarán P3 y la Rut(S,P1), y en bloqueado sigue quedando P1.

Supóngase que en el ínterin P2 hace cosas. No importa qué es lo que hace. Pero en algún momento aparece el planificador de procesos que elige a P3 para que se ejecute, entonces ahora queda la Rut(S,P1) en la cola de listos y en bloqueado quedan P1 y P2.

Ahora supóngase que P3 pide exactamente lo mismo que P1, o sea que P3 hace la llamada al Supervisor, pidiendo el recurso S, nuevamente se tendrá una rutina que atiende la llamada (At.Sup.), cuando sucede esto ahora en la zona de interrupciones no se va a tener más la palabra de control de P1, sino que se tendrá la de P3.

Ahora es P3 quien tiene que pasar al estado de bloqueado, el sistema tiene que armar una rutina para el recurso S pero ahora para el proceso P3, y nuevamente se llama al controlador de tráfico. Mientras tanto en la cola de listos y en bloqueados siguen los mismos estados.

Una vez que se llama al controlador de tráfico este va a poner en la cola de listos Rut(S.P3) y pondrá en el estado de bloqueado a P3.

Después va a llamar otra vez al planificador de procesos y éste va a elegir de acuerdo al orden que tenían las cosas a la rutina que pide S para P1 (Rut(S,P1)), en la cola de listos tenemos la Rut(S,P3) y tenemos bloqueados a P1,P2 y a P3.

Esta rutina de alguna manera invoca, o tiene dentro de sí el pedido de asignación del recurso S, durante esa situación se va a hacer P(X), luego X = X - 1, si X<<0 se pasa a un Wait, digamos que inicialmente el valor del semáforo es 1, por lo tanto X va a valer 0 y se asigna el recurso al proceso 1.

Asignárselo al P1 significa que ahora el proceso lo puede usar. Supóngase que lo que está pidiendo es una porción de memoria, si está listo para usarla, hay que hacer ciertas cosas como sacarlo del estado de bloqueado en que se encuentra y pasarlo a la cola de listos para que la pueda usar. Entonces se tiene que llamar al controlador de tráfico para el proceso P1, informándole que lo saque del estado de bloqueado. En conclusión, se tendrá a la rutina que pide S para P3, luego, se tiene en este momento el P1 que acaba de salir del estado de bloqueado y pasa a la cola de listos y P2 y P3 continúan bloqueados.

Cuando termine el controlador de tráfico de ordenar todas sus listas, llama otra vez al Planificador de procesos, y el planificador de procesos seleccionará la rutina que pide el recurso S para P3, en la cola de listos estará P1 y en bloqueados P2 y P3.

Entonces se invocará nuevamente la rutina de asignación, va a quedar en (-1) el valor del semáforo y pasará a un WAIT, y como ya se ha dicho pasar a un WAIT es pasar al estado de bloqueado por semáforo, entonces la primitiva WAIT coloca en el estado de bloqueado del semáforo X a esa rutina del controlador de tráfico, siempre es el controlador de tráfico el que toca las tablas, y la situación queda con que P1 está listo, P2 y P3 también bloqueados y se agrega en bloqueado por semáforo a Rut(S,P3).

En realidad a pesar de que el recurso no estaba siendo usado, no se lo otorgó porque había sido asignado a otro proceso.

Luego seguirá el planificador de procesos como corresponde y finalmente P1 hará uso de su recurso. El resto es trivial.

La rutina S al asignar, le da el puntero. Si fuera un buffer de memoria entonces le dice cuál es la zona de memoria que le está asignando, le da la dirección del lugar donde puede escribir, por ejemplo. En el caso de operaciones de E/S que asigna el canal significa que está poniendo al proceso en la posición necesaria para que lo use.

EJECUCION	**LISTOS**	**BLOQUEADOS**	**BLOQ. POR SEMAFORO**
P1	P2 P3		
P1 (Sup S)	P2 P3		
Atenc. sup.	P2 P3		
(P1 Bloq)			
Rut (S,P1)			
Llamar CT			
CT	P2 P3		
CT	P2 P3 Rut(S,P1)	P1	
PP	P2 P3 Rut(S,P1)	P1	

P2	P3 Rut(S,P1)	P1	
.....	......	....	
PP		P1 P2	
P3	Rut(S,P1)	P1 P2	
P3 (Sup S)	Rut(S,P1)	P1 P2	
At. Sup.	Rut(S,P1)	P1 P2	
(P3 Bloq)	Rut(S,P1)	P1 P2	
Rut(S,P3)	Rut(S,P1)	P1 P2	
Llamar CT	Rut(S,P1)	P1 P2	
CT	Rut(S,P1) Rut(S,P3)	P1 P2 P3	
PP			
Rut(S,P1)	Rut(S,P3)	P1 P2 P3	
CT(P1)	Rut(S,P3) P1	P2 P3	
PP			
Rut(S,P3)	P1	P2 P3	
CT	P1	P2 P3	Rut(S,P3)
PP			
P1		P2 P3	Rut(S,P3)

Fig. 12.27.

CAPITULO 13

ADMINISTRACIÓN DE MEMORIA

13.1 - **INTRODUCCIÓN**

Es bien sabido que tanto el Procesador como los dispositivos de E/S interactúan con la Memoria. Así también un programa debe estar en Memoria para poder ser ejecutado ya que el procesador levanta de allí las instrucciones que debe ejecutar.

Ahora bien si se quiere compartir la memoria entre varios programas, por ejemplo en un entorno de multiprogramación, y permitir además el correcto funcionamiento de los canales de E/S, se hace necesario administrar la memoria entre tales programas.

De esto último se deduce que el Administrador de Memoria es activado toda vez que se carga un trabajo. Luego veremos que no solo se activa al cargar un trabajo sino que también tiene otras funciones.

Los métodos de administración de memoria han ido evolucionando, lógicamente desde el más sencillo creciendo en complejidad; seguiremos pues ese orden.

Para todos los casos estudiados, salvo que se indique lo contrario, se supondrán memorias de direccionamiento continuo.

13.2 - **Administración de Memoria SIMPLE CONTIGUA**

En este sistema de administración la memoria aparece al programa como una única extensión contigua de direcciones, compartida solo por él y por el sistema operativo.

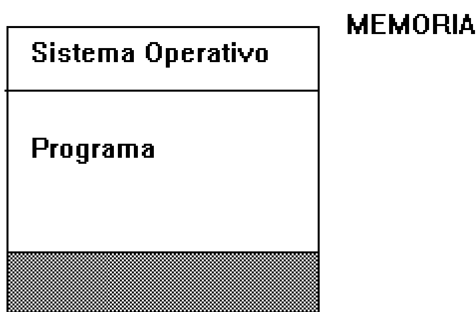


Fig. 13.1. - Administración de Memoria Simple Contigua.

Se hace necesario lograr en este tipo de administración algún mecanismo de protección para el sistema operativo que es residente.

Existen dos alternativas:

1) la utilización de un límite dado por un registro fijo por hardware o registro de protección cargado por el sistema operativo que sirve como punto de comienzo de carga del programa como en el Fortran Monitoring System IBM 3094 o como límite como en el caso de la H.P. 2116B que carga desde memoria baja. Su desventaja es que requiere la recompilación de los programas al modificarse el valor de tal registro.

2) otra opción consiste en un registro Límite o registro Base (o de Reubicación) (tal el caso de la CDC 6000).

Mediante alguno de estos dos sistemas el sistema operativo intercepta aquellas direcciones inválidas generándose entonces una interrupción por error de direccionamiento.

Por tanto esta administración requiere desde el punto de vista del **hardware** los elementos para proteger al sistema operativo, y desde el punto de vista del **software** las facilidades al programa (E/S) y la rutina de atención de interrupciones por invasión al sistema operativo.

Sus desventajas son :

- el desperdicio de recursos tanto de la memoria no utilizada, como también el desperdicio respecto de los periféricos no usados durante la ejecución del programa.
- la imposibilidad de ejecutar programas que requieran el uso de más memoria que la disponible, por más que el sistema posea una capacidad de direccionamiento mayor.

13.3 - **Capacidad de direccionamiento vs. Capacidad de memoria**

Por un lado se habla de memoria disponible y por otro de la capacidad de direccionamiento del sistema. El punto es el siguiente : si el direccionamiento de un sistema está dado por 24 bits esto implica que es posible direccionar hasta 16 Mbytes de memoria y no más.

Pero si la memoria disponible en este sistema es de 3 Mbytes, el tamaño máximo de programa que pueda residir en este sistema será de 3 Mbytes (descontando el espacio correspondiente al Sistema Operativo) y **no** más, aunque la capacidad de direccionamiento sea de 16 Mbytes.

Este tipo de inconvenientes se encontrará en todo sistema de Administración de Memoria que requiera que la totalidad del programa resida en memoria para poder ser ejecutado.

13.4 - **Soluciones a la monoprogramación**

Para resolver el problema de la residencia de un único programa en memoria que conlleva monoprogramación, y la limitación de memoria, tanto sea por su escasez como por su limitación de direccionamiento, se han utilizado algunos artilugios, de los cuales veremos un par.

13.4.1 - **Simulación de multiprogramación ó Swapping**

Esta técnica consiste en aprovechar los tiempos en los cuales un programa (proceso) se encuentra ocioso (debido a que un operador está pensando, como en el caso de un editor, o está realizando una operación de E/S sobre un periférico lento) para desalojarlo completamente de la memoria, sobre un periférico rápido, y cargar desde un periférico similar otro programa que estuviese pronto para su ejecución.

Obviamente esta técnica implica la necesidad de preservar buffers de memoria para recibir/emitir datos de

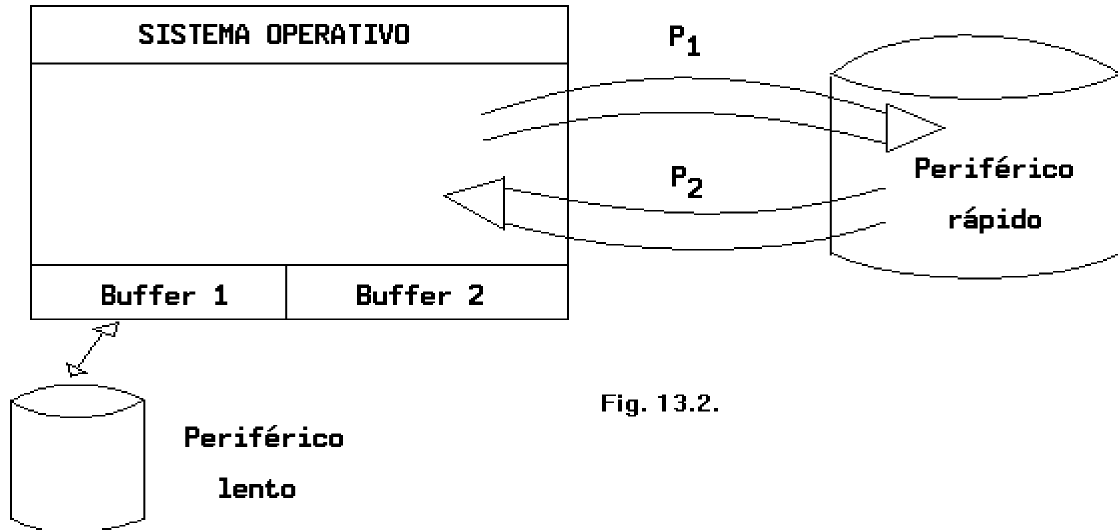


Fig. 13.2.

entrada/salida para cada proceso que conviva en un determinado momento dentro del sistema. Además la mezcla de procesos debe ser tal que ninguno sea de alto uso de procesador, pues, sino esta técnica no resultaría útil. Son necesarios además, elementos que controlen las operaciones de E/S, para permitir que el procesador pueda dedicarse a la ejecución de programas.

Esta técnica fue utilizada por el Bull 61 con cierto éxito, mientras el número de procesos fuese pequeño (4) y ninguno de alto consumo de procesador.

Actualmente esta técnica es utilizada también en sistemas operativos que manejan multiprogramación, inclusive en Arquitecturas de tipo MIMD cuando, por algoritmo, se decide castigar a un proceso por haber excedido la utilización de algún recurso (en particular el procesador, por ejemplo en la IBM OS/MVS).

13.4.2 - **Simulación de mayor direccionamiento a memoria ó Técnica de Overlay**

Esta técnica consiste en aprovechar la modularidad de los programas y en particular en la capacidad de detectar conjuntos de módulos o subrutinas que serán ejecutados en forma disjunta.

Normalmente se confecciona y vincula un programa y sus subrutinas en forma lineal, o sea dándole a cada uno de ellos su propia dirección, como por ejemplo puede verse en la Fig. 13.3, programa que no podrá ser ejecutado en una memoria de capacidad 400.

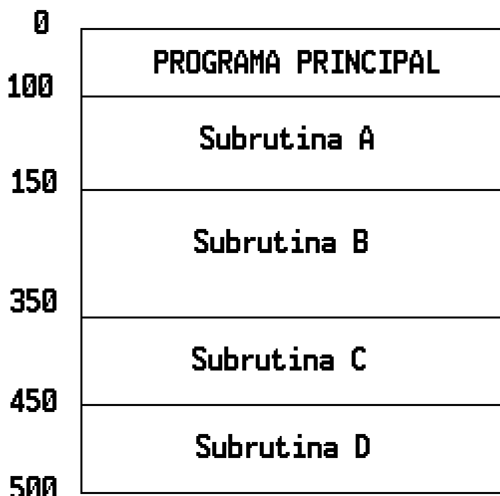


Fig. 13.3.

Ahora bien, si el programa pudiese ser descompuesto en forma lógica de la manera que se ve en la Fig. 13.4, o sea que cuando se utilicen las subrutinas SubA y SubB (rama A), no se utilizan las SubC y SubD (rama B), es decir, que el uso de la rama A es excluyente con la rama B, podría pensarse y vincularse a este programa reutilizando direcciones, o sea que podría guardarse en una biblioteca con el direccionamiento en la forma que se muestra en la Fig. 13.5 lo que permitiría que en algún momento se estuviese ejecutando la rama A

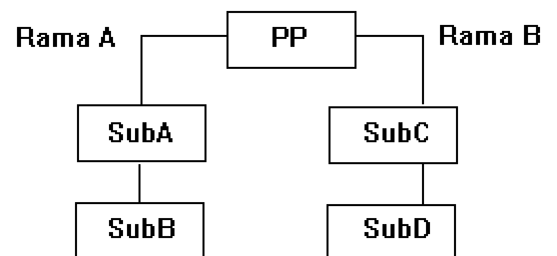


Fig. 13.4.

con un mapa de memoria como se ve en la Fig. 13.6 a) y en otro la rama B con un mapa de memoria como se ve en la Fig. 13.6 b), con lo cual ahora sí es posible ejecutar este programa en una memoria de capacidad 400.

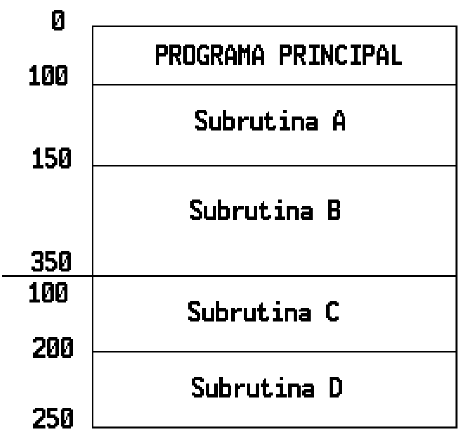


Fig. 13.5.

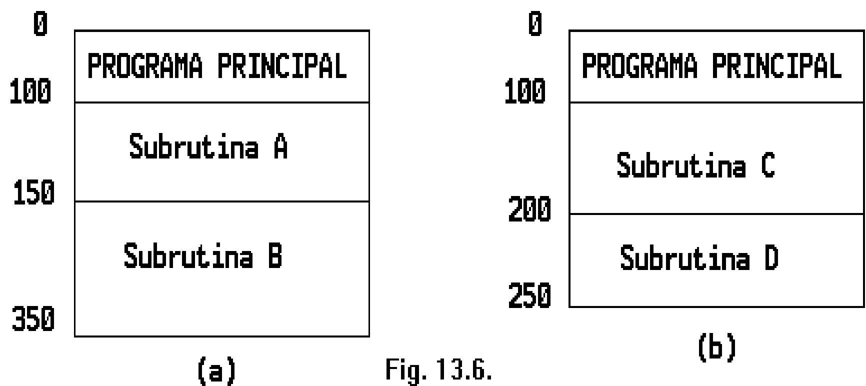


Fig. 13.6.

Esta estructura de overlay se puede pensar en forma estática, o sea previa a la carga del programa se determinará qué rama se ejecutará, y durante toda esa sesión se ejecutará esa sola; o en forma dinámica, en la cual la lógica del programa PP será quien determine qué rama se carga y si está cargada la rama A y se necesita la rama B, esta "tapará" a la rama A y será ejecutada, y así sucesivamente.

13.5 - **Administración de memoria PARTICIONADA FIJA**

En este esquema se establecen particiones fijas de la memoria de una sola vez y para siempre (por hardware o por sistema operativo), o en caso contrario esas particiones son cambiables en tamaño mientras no haya trabajos ejecutándose (normalmente esta tarea la realizará el operador desde consola).

La protección aquí se logra mediante un registro Base (también de reubicación) y un registro Límite o Longitud asociados a cada uno de los trabajos que se están ejecutando.

En este método aparece una fragmentación de la memoria debido a que parte de la partición no es utilizada por el programa.

Los mecanismos de protección y reubicación mediante los registros Base y Longitud son mecanismos **hardware** que surgen en este método de administración. Se agregan las interrupciones por direccionamiento (intento de acceder fuera de la partición).

En cuanto a **software** tenemos las rutinas de atención de interrupciones, las rutinas que se encargan de manejar las tablas de dicha administración, rutinas para asignación de archivos y dispositivos y los programas de canal.

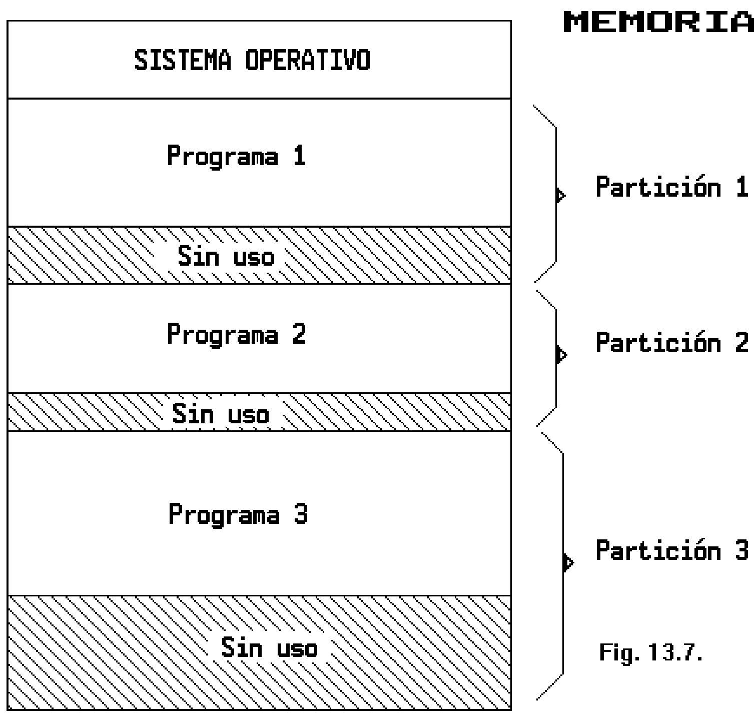


Fig. 13.7.

Las tablas necesarias para manejar este tipo de administración son las siguientes :

- Una tabla de tamaños (una sola en el sistema) que es de la siguiente forma :

PROCESO	DIRECCION	LONGITUD	ESTADO
P1			ejecutando
P2			bloqueado
P3			listo

Fig. 13.8.

- Un Bloque de Control de Proceso (BCP, uno para cada proceso que se encuentre dentro del sistema) que debe contener :

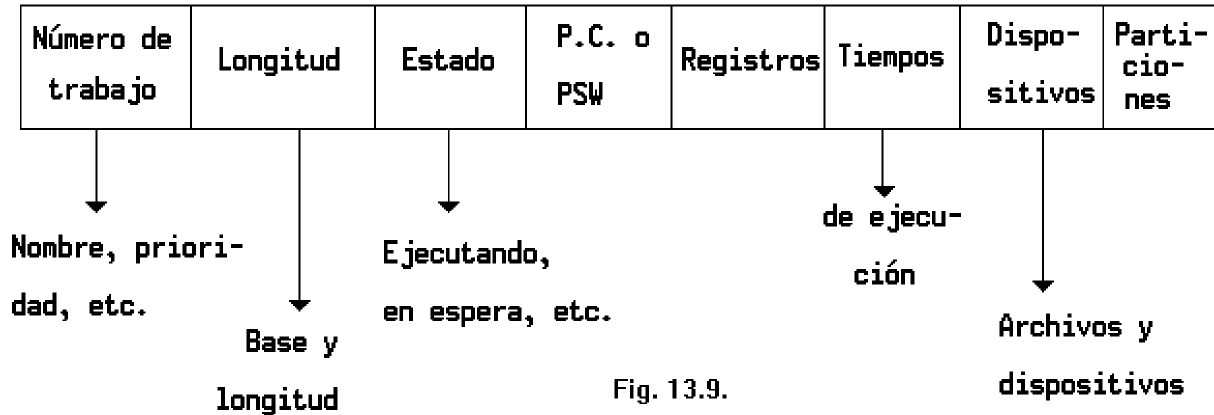


Fig. 13.9.

Obviamente este sistema de Administración de Memoria es el que comienza a permitir la ejecución en multiprogramación. Uno de los primeros existentes fue el IBM OS/360 MFT.

13.6 - **Administración PARTICIONADA VARIABLE SIN COMPACTACION.**

En este esquema las particiones se establecen según la longitud de los programas iniciales.

Al cabo de un tiempo se produce mucha fragmentación, por tanto existen diversas políticas para asignar una partición de memoria libre.

En principio sea cual fuere el mecanismo de asignación de una partición libre se hace necesario contar con la información de cuáles son esas particiones libres de memoria lo cual nos lleva a la aparición en este tipo de administración de una lista de zonas disponibles.

Existen tres grandes políticas de asignación :

1) **La primer zona libre** : aquí se recorre la tabla de espacios libres y la primer partición que se encuentre en donde quepa el programa es asignada.

2) **La mejor zona** : se recorre la tabla buscando aquella partición que mejor se ajuste al programa que se desea ingresar.

3) **La peor zona** : se recorre la tabla buscando la mayor partición libre existente, la que será asignada al nuevo proceso.

El **hardware** necesitado en esta administración coincide con el de particionada fija, en tanto que desde el punto de vista del **software** nuevo se agregan aquí las nuevas rutinas de manejo de las tablas de zonas libres para el mecanismo de asignación.

Uno de los primeros sistemas operativos que utilizó este esquema fue el IBM OS/360 MVT.

13.7 - **Administración PARTICIONADA VARIABLE CON COMPACTACION.**

Cuando la cantidad de memoria fragmentada es tal que su suma permite el ingreso de un trabajo nuevo (el cual de otra forma no podría ingresar al sistema) se "compactan" los trabajos que se encuentran en memoria.

Para realizar esta compactación es necesario contar con el registro de reubicación ya que de otra forma al mover un programa de su posición original en memoria deberían alterarse las instrucciones de tal programa.

Se puede realizar una compactación directamente sobre memoria (Memoria-a-Memoria) o utilizando dispositivos periféricos (Memoria-a-Disco-a-Memoria), en este último caso se descarga bastante trabajo de la CPU, ya que la misma puede quedar eventualmente libre para continuar la ejecución de alguno de los procesos que no se encuentran involucrados en la compactación. En el caso de la compactación Memoria-a-Memoria la CPU se encuentra dedicada a full a la tarea de compactación, por lo tanto se detienen las ejecuciones de **todos** los procesos.

Existen diversas técnicas de compactación pero ya que el mecanismo de compactación es sumamente costoso se trata de disminuir lo más posible la cantidad de veces que se compacta como así también se intenta compactar en aquellos casos en que la cantidad de bytes involucrados sea mínima.

Nuevamente aparecen aquí los mecanismos **hardware** de protección y reubicación, y se agregan como mecanismos de **software** las rutinas de compactación y reubicación y desde ya tienen las mismas tablas que indican la ubicación de los trabajos y las zonas libres.

Algunos sistemas operativos que los utilizaron fueron el GECOS del BULL 66 y el SCOPE de la CDC 6600.

13.8 - **Administración de memoria PAGINADA**.

En este esquema la memoria física se divide en Bloques de igual tamaño, cada uno de los cuales contendrá una Página de programa. Las páginas de los programas estarán ubicadas en los bloques de memoria los cuales no tienen porque ser contiguos. (Nótese esta fundamental diferencia frente a las otras Administraciones de Memoria).

Al ser cargado en proceso a memoria se crea una Tabla de Distribución de Páginas (TDP) que está apuntada desde el Bloque de Control de Proceso (BCP) y cada una de sus entradas indican en qué bloque de memoria real se encuentra cargada cada una de las páginas del programa.

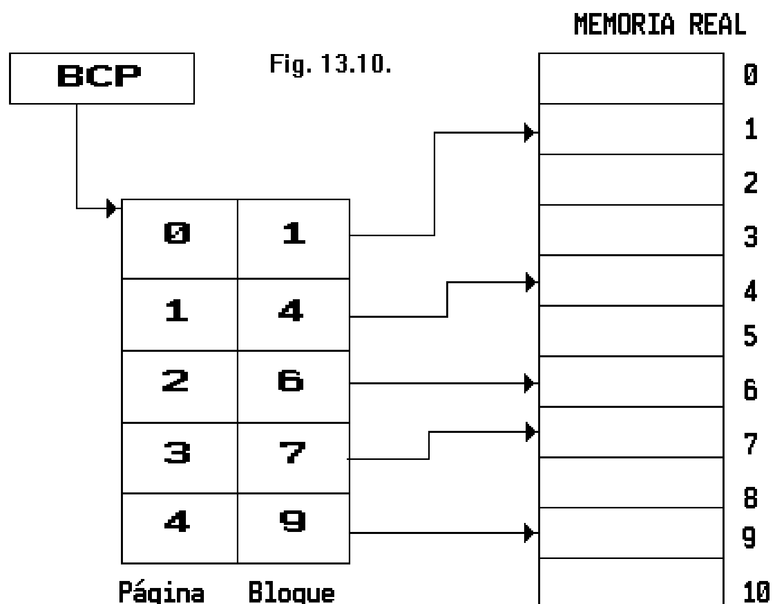


Fig. 13.10.

13.8.1 - **Cálculo de la dirección**

Las direcciones lógicas están compuestas de dos partes :

Número de Página	Desplazamiento dentro de la Página	Dir. del Programa.

Fig. 13.11.

El número de página es tomado por el DAT (Direct Address Translator) el cual accede a la TDP y obtiene el número de bloque que corresponde a esa página para ese proceso y obtiene :

Número de Bloque	Desplazamiento dentro de la Página	Dir. en Mem. Real

Fig. 13.12.

Por ejemplo :

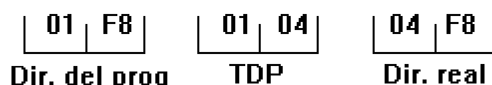


Fig. 13.13.

Las tablas están en memoria, luego para obtener una dirección efectiva a cualquier posición de un programa se torna necesario acceder dos veces a memoria : la primera para llegar a la TDP y obtener la dirección del bloque correspondiente y la segunda para llegar efectivamente a la dirección deseada.

Una de las soluciones posibles a este doble acceso es cargar la TDP en el procesador en registros (se cargarían juntamente con la PC y los registros en el momento de cambio de contexto), pero esto limita mucho la cantidad de páginas que puede tener un programa y es además muy costoso.

Una solución más económica es el uso en memoria CACHE que contiene todas o una porción de las entradas de la tabla. Por el mismo algoritmo de actualización de la cache nos aseguraríamos que con el transcurrir del tiempo las entradas que permanecerán en cache serán las más usadas.

Como mecanismos **hardware** nuevos se agregan el Traductor de Direcciones (DAT), los registros de páginas o la memoria Cache y como mecanismo de protección se suele utilizar un bit (o clave) de protección asociado a cada bloque.

La clave de protección será usado para todos los accesos a memoria que no se realicen por medio del DAT.

Como rutinas de **software** se agregan en esta administración las tablas de páginas por programas y las rutinas de manejo de dichas tablas.

13.9 - **MEMORIA VIRTUAL**

Definimos como **Espacio de Direccionamiento de un Sistema** (computadora) a la capacidad de direccionamiento de la misma, o sea la dirección a la que puede llegar con todos los bits de direccionamiento en ON, por ejemplo de 0 a mm.

Definimos como **Espacio de Direccionamiento de un Proceso** a la capacidad de direccionamiento del mismo, o sea que pueda generar o pedir direcciones de 0 a nn.

Obviamente para que este proceso ejecute en este sistema debe ser nn £ mm.

Por otra parte ya sabemos que no necesariamente la memoria disponible en una computadora es igual a su espacio de direcciones, sino que generalmente es más chica, e inclusive puede ser menor que el espacio de direccionamiento requerido por un proceso.

Muchas veces es necesario escribir programas cuyo tamaño supere el tamaño de la memoria real existente en la instalación.

A raíz de esta necesidad surge lo que denominaremos MEMORIA VIRTUAL que simula poseer una cantidad de memoria mayor que la existente, y a continuación se tratan los métodos de administración para este tipo de memorias.

Usualmente en Memoria Virtual la memoria aparece al programador tan grande como él la necesite, existiendo en este caso un mecanismo provisto por el método de administración que se encarga de poner a disposición del procesador los fragmentos (tanto páginas como segmentos) que sean requeridos en el momento de que sucede tal requerimiento.

Se debe tener en cuenta que si un proceso utiliza para si todo el espacio de direcciones de un sistema, mientras él esté ejecutando será el único proceso en ejecución (lo que comúnmente se llama Espacio único de direcciones).

Ahora bien, si se puede simular, en las técnicas que se verán más adelante, la posibilidad de poseer mayor memoria real que la que existe, porqué no es posible simular también que existen más espacios de direcciones ? (lo que comúnmente se llama Espacio múltiple de direcciones).

Esto se puede lograr independizando completamente las direcciones de los procesos de las direcciones de memoria real, permitiendo que cada proceso posea su propio juego de direcciones, o sea permitiendo que existan tantas direcciones 00 ó 100 como procesos existan. Esta es una conclusión casi trivial cuando se observa la construcción de las Tablas de Distribución de Páginas.

13.10 - **Administración PAGINADA POR DEMANDA**

Para ver el funcionamiento de este tipo de administración debemos contar primero como llega un programa al sistema.

Originalmente el programa se encuentra en una biblioteca (usualmente en algún periférico tipo disco magnético) desde la cual es invocado para su ejecución.

El programa se carga entonces en otro periférico (aquí también usualmente discos magnéticos) en donde es preparado para la ejecución (se arma su BCP, se completa su TDP, se le agrega información respecto a buffers de archivos, etc.).

Una vez que el programa está listo se carga su TDP en memoria real y se pasa al proceso a estado de Listo en espera del control de la CPU.

Una peculiaridad que existe en esta administración es que en la TDP se agrega un bit que indica si la página que se está referenciando existe en memoria real o no y además se agrega un campo que indica la dirección de tal pagina en el dispositivo de memoria virtual.

Los programas con el transcurrir del tiempo no se cargan en su totalidad en memoria real, sino que se cargan aquellas páginas que van siendo solicitadas para su ejecución.

Para su administración se necesitan un par de tablas cuyos contenidos iremos analizando.

Tabla de Distribución de Páginas (una para cada proceso)

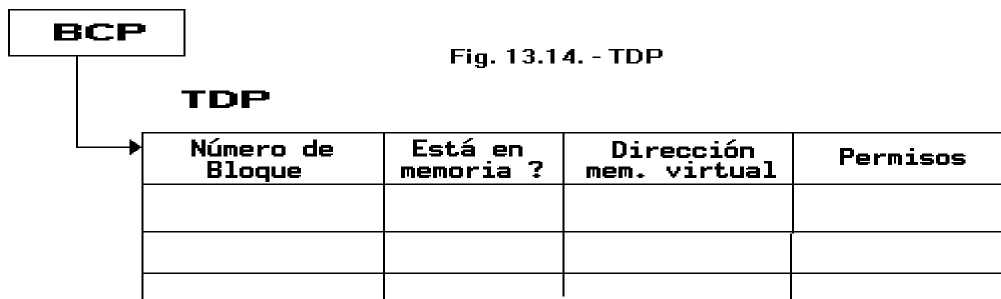


Tabla de Distribución de Bloques (una en todo el sistema)

Cuando se encuentra encendido el bit que indica que la página no se encuentra en memoria real se produce una interrupción por falta de página. La rutina que atiende esa interrupción necesita la dirección de memoria

Fig. 13.15. - TDB

TDB

Id. del programa	Página	Contador	Página cambió ?	Fijo por canal ?	Tránsito?

virtual en donde se encuentra almacenada dicha página para poder traerla a memoria real.

Supongamos que la dirección del lugar en donde se encuentra la página en memoria virtual se encuentra a nivel del BCP en lugar de figurar en la TDP para cada una de las páginas que componen ese programa, es decir que existe una única dirección que apunta a donde se encuentran almacenadas en memoria virtual las páginas de este programa.

La conclusión inmediata es que esta forma nos acarreará un problema de administración respecto de la contigüidad del programa en memoria virtual y además hará sumamente engorroso el ubicar una página determinada ya que llevaría a la lectura secuencial de todas sus páginas (comenzando desde la dirección apuntada desde el BCP).

La otra manera, o sea, cada entrada en la TDP tiene su propio apuntador a memoria virtual, permite mantener en forma discontigua a los programas en disco y acceder en forma directa a la página buscada.

El bit de cambio que se encuentra en la TDB es necesario para toda vez que sea imprescindible remover una página de memoria real con el objeto de traer otra, ya que si la página que se va a remover sufrió modificaciones debe ser regrabada en memoria virtual para preservar la integridad de la aplicación.

El bit de Fijo por canal se utiliza para fijar páginas en memoria real; esto es, cuando una página contiene información que está siendo dirigida o está recibiendo información de un canal de E/S, debido a que los canales deben utilizar direcciones absolutas se torna imprescindible fijar tales páginas en memoria real. Es decir no pueden ser removidas. Nótese que también se encuentran en esta situación aquellos bloques de memoria real sobre los cuales se está cargando una página o de los cuales se está descargando una página en memoria virtual. Normalmente a estas últimos bloques suele denominárselos "en tránsito".

13.10.1 - **Interrupción por PAGE FAULT.**

El mecanismo de direccionamiento opera igual que en el método de administración de memoria paginada simple.

Si el procesador intenta direccionar a una página que no se encuentra cargada en memoria real se produce lo que se denomina una **Interrupción por falla de página (Page-Fault)**.

Podemos graficar el mecanismo completo de direccionamiento incluyendo el procesamiento de la interrupción de la falla de página como se puede visualizar en la Fig. 13.16.

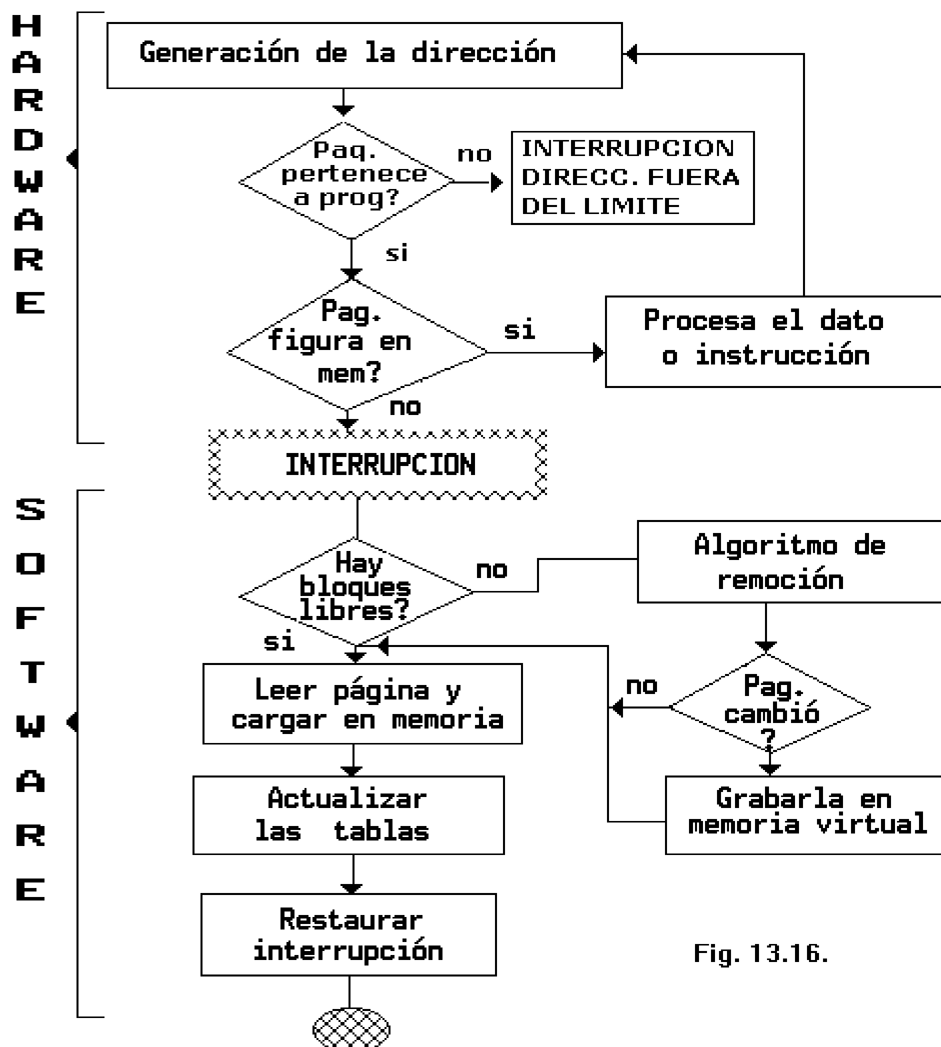


Fig. 13.16.

Desde la generación de la dirección hasta la interrupción inclusive, todos los pasos se realizan por **hardware**. En cambio desde el momento en que se comienza a ejecutar la rutina que atiende esa interrupción hasta la restauración final del estado del estado previo antes de la interrupción se realiza por **software**.

Si la interrupción se produce a causa de una dirección generada por el RPI no hay mayor problema, pues no se había comenzado la ejecución de la instrucción.

En cambio, si la interrupción se produce como consecuencia de una dirección generada por los operandos de una instrucción que está ejecutándose se nos plantea un problema ya que nos encontramos en medio de la ejecución.

Existen dos caminos posibles :

- Recomenzar (restart) : Al producirse la interrupción se restaura la instrucción al estado inicial (previo a su ejecución). Hay que volver también los registros y posiciones de memoria cambiados a su estado anterior, por lo tanto esto nos obliga a llevar una historia de la instrucción. Una implementación sencilla que nos ahorraría el mantener esta historia sería hacer antes que nada la verificación de las direcciones de los operandos (o sea antes de modificar nada) (Esta técnica acompaña muy frecuentemente a un Pipeline).
- Continuación : Se salva el estado de la instrucción y luego de satisfecha la interrupción se continúa la instrucción en el punto en donde se había abandonado. Esto implica salvar la información a nivel de micro-instrucción (salvar registros temporarios, estado de la instrucción, etc.). No todo tipo de instrucción puede soportar esto (pensemos por ejemplo en la instrucción Test & Set), ya que se ven obligadas a que en caso de ser interrumpidas a Recomenzar, por lo tanto se hace necesario el aplicar distintos mecanismos según el tipo de instrucción.

13.10.2 - **Traza**

Definiremos como **traza** de un programa a la enumeración de las páginas que ese programa referencia a medida que se ejecuta.

Definiremos también la siguiente relación :

f = número de páginas traídas / número de referencias de la traza

siendo f el **Indice de Fracasos** y si calculamos

s = 1 - f

llamamos a s el **Indice de Hallazgos**.

Lógicamente estos valores sufren una sustancial modificación dependiendo de los bloques libres que existen en memoria real sobre los cuales se pueden cargar páginas desde memoria virtual.

Es por tanto necesario antes de calcular "s" o "f" conocer la cantidad de bloques libres en memoria real.

La carga de la primer página de la traza implica también un fracaso ya que tanto si es invocada por el programa al ejecutar su primera instrucción o por que el Planificador de Trabajos le carga la primer página al proceso a ejecutar ambas situaciones implican una falla de página.

13.10.3 - **Algoritmos de Remoción**.

Llegado el momento que la tabla de distribución de bloques indica que no existe ningún bloque libre para asignar a una página que se está solicitando se hace necesario por tanto eliminar una página de memoria real. Denominamos a esto **Remoción**, y existen diversas políticas de selección de la víctima.

El mecanismo **óptimo** sería seleccionar aquella página que se tardará más en volver a usar, pero lamentablemente esto es sumamente difícil de predecir.

Es entonces a partir de aquí que se comenzará a utilizar el campo Contador de la TDB.

Otro mecanismo es el **FIFO** (first-in first-out), en donde la página que se selecciona para remoción es aquella más antigua en el sistema.

El método FIFO produce en determinados casos lo que se denomina "Anomalía de Belady". Supongamos un trazado del siguiente tipo :

1 2 3 4 1 2 5 1 2 3 4 5

si se realiza el cálculo de los indices de hallazgos y de fracasos para un tamaño de memoria de

M = 3 bloques y

M = 4 bloques

se obtiene que en el caso en que se tienen 4 bloques libres (es decir que se ha incrementado la cantidad de memoria libre que se puede asignar) se obtiene un índice de fracasos mayor que en el caso de M = 3.

Como conclusión se puede decir que a partir de un determinado punto el incrementar la memoria disponible no mejora el índice de fracasos e inclusive en determinados casos hasta lo empeora.

Una mejora que se puede introducir al algoritmo FIFO es el de otorgar una **segunda chance**, mecanismo mediante el cual si la página es referenciada una segunda vez se pone a cero un bit que la acompaña. Por tanto esa página tiene otra oportunidad para permanecer en memoria y no ser seleccionada para remoción.

Otro método de remoción es el **LRU (Least Recently Used)**. En este método aquella página que ha sido menos recientemente utilizada es la que se selecciona para remoción.

Lógicamente para lograr este fin el sistema debe llevar la información necesaria para poder determinar quién es la víctima. Esto se logra mediante bits que se colocan a cero o a uno dependiendo de cuanto tiempo hace que la página fue referenciada.

Otro algoritmo es el **LFU (Least Frequently Used)**, en este caso la página menos usada es la elegida para remoción. Para poder lograr esto el sistema lleva una cuenta sobre cuantas veces la página fue referenciada.

Otro método consiste en no solo llevar información respecto a las referencias realizadas a la página, sino también utilizar la información del bit de cambio, ya que en aquellos casos en que la página que se selecciona para remoción ha sido alterada es absolutamente imprescindible su regrabación en memoria virtual.

13.10.4 - **Cuestiones de implementación**.

En general el tamaño de la página coincide exactamente con el tamaño del bloque, aunque existen implementaciones en las que el tamaño de la página es un múltiplo del tamaño del bloque.

A efectos de simplificar nuestro trabajo asumiremos en nuestra materia que el tamaño de la página coincide siempre con el tamaño del bloque.

En este sistema de administración de memoria existe fragmentación, ya que generalmente la última página del programa no ocupa exactamente un bloque. Sin embargo la fragmentación solo llega a lo sumo al tamaño de una página por cada programa dentro del sistema.

Un problema bastante frecuente en esta implementación de administración de memoria es el buffer que cruza el límite de una página a otra, usualmente denominado "buffer a caballo".

Afortunadamente los canales pueden manejar esta eventualidad utilizando para ello programas de canal concatenados que permiten transferir parte de la información a un sector de memoria real y continuar la transferencia sobre otras direcciones de memoria. Recuérdese que en estos casos ambas páginas deben estar fijas en memoria.

Cuando el algoritmo de remoción es muy malo o en aquellos casos en que la memoria real es drásticamente más pequeña que la memoria virtual se puede producir un fenómeno denominado **thrashing**.

Thrashing es el estado en que se encuentra un sistema cuando esta realizando un excesivo intercambio de páginas, en otras palabras, el sistema pasa más tiempo ejecutando las rutinas de remoción y carga de páginas que en ejecutar las instrucciones de los procesos propiamente dichos.

Como mecanismos nuevos de **hardware** se agregan en este tipo de administración : los mecanismos de protección por bloques (que no son utilizados por el DAT), la interrupción por falla de página, los contadores de uso de las páginas, los bits de páginas cambiadas y la fijación de páginas por el canal de E/S.

Como elementos nuevos de **software** tenemos aquí la Tabla de Distribución de Páginas, las rutinas de atención de interrupciones por falla de página, los algoritmos de remoción, la rutina de búsqueda de páginas, la de grabación de páginas, las rutinas de actualización de las tablas en memoria y las rutinas de administración de la memoria virtual.

13.11 - **Administración de Memoria con SEGMENTACION**.

Número de segmento	Desplazamiento en el segmento

Fig. 13.17.

La administración de memoria segmentada puede darse con o sin memoria virtual, aquí la veremos con memoria virtual.

En este esquema la división de la memoria se produce según el tamaño de los segmentos.

Un **segmento** es una unidad lógica del programa; por ejemplo : una división lógica, un área Common (en Fortran), un vector, una Section del Cobol, una subrutina, etc.

Necesitamos también tablas de administración de segmentos que son mayores que las tablas de páginas ya que aquí los segmentos difieren en longitud, y por tanto debemos guardar para cada segmento no solo su ubicación en memoria sino su tamaño. (Implementado en equipos tales como el Burroughs B5500 y el Honeywell

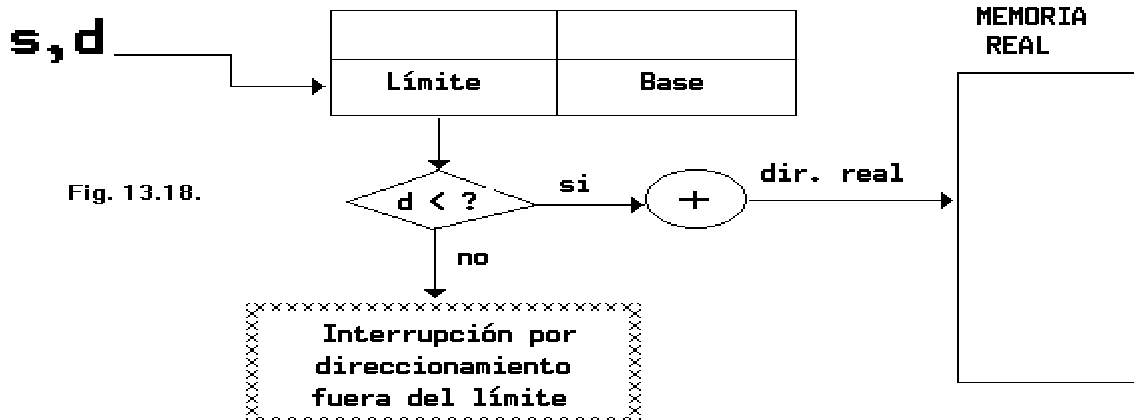


Fig. 13.18.

6180 bajo el sistema operativo MULTICS).

Cada dirección se encuentra desdoblada en número de segmento y desplazamiento (offset) dentro del segmento (S,d) :

Ya que para cada segmento se tiene su dirección de origen (base) y su longitud (límite), el mecanismo de direccionamiento opera de la forma que puede visualizarse en la Fig. 13.18.

Los segmentos también pueden ser compartidos, para lo cual deben ser reentrantes y existen además permisos de acceso asociados a ellos.

Tales segmentos se vinculan solamente en el momento de ser necesarios y tal vinculación se realiza en memoria virtual. Esto nos lleva a una característica de este tipo de administración de memoria que se denomina "**vinculación dinámica"**.

Cuando un programa se compila en este tipo de sistemas de administración no se resuelven todas sus direcciones en tiempo de vinculación, ya que de existir referencias externas estas se resuelven solamente en tiempo de ejecución. Esto permite justamente el poder referenciar a segmentos compartidos por varios usuarios y no duplicar innecesariamente copias del mismo en memoria real.

Existe fragmentación también en este tipo de administración debido a las áreas libres que quedan en memoria real y que no pueden ser asignadas a ningún segmento, un poco como la situación planteada en la administración Particionada Variable. Y aquí también puede existir algún mecanismo de Compactación.

13.11.1 - **Tablas necesarias**.

Las tablas que se requieren en este tipo de administración son cuatro, a saber :
- Tabla de Mapa de Segmentos (existe una por cada espacio de dirección).
- Tabla de Areas no Asignadas (existe una sola en el sistema).
- Tabla de Nombres de Segmentos (existe una por cada espacio de dirección).
- Tabla de Estado de Segmentos Activos (una en todo el sistema).

Una implementación posible de estas tablas puede ser visualizada como se ve en la Fig. 13.19, por razones de simplicidad pondremos a las tablas de Mapas de Segmentos y Nombres en una sola.

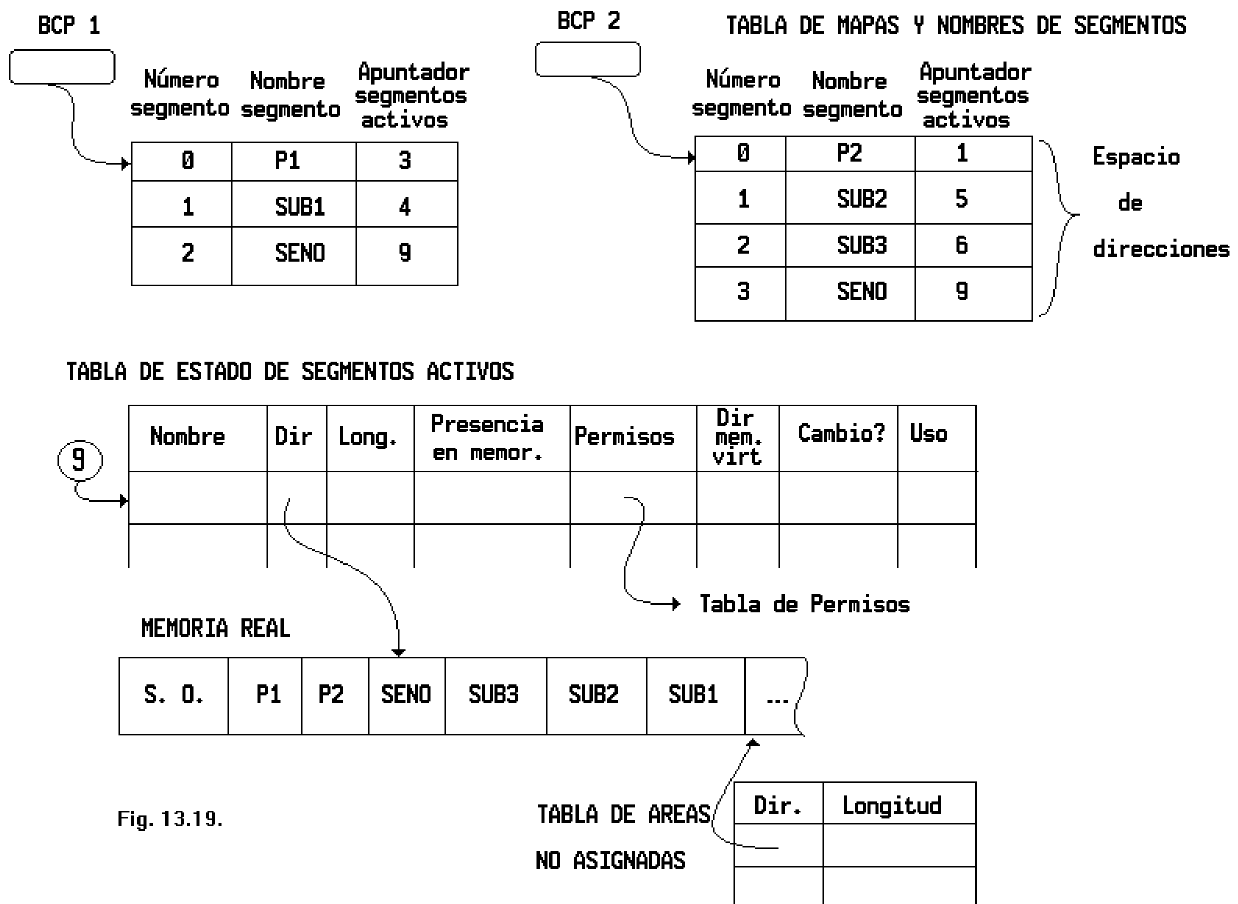


Fig. 13.19.

13.11.1.1 - **Mecanismo de trabajo de las tablas**

El mecanismo de trabajo de las tablas sería, por ejemplo, que si desde el proceso P2 se solicita la utilización de la subrutina Sub2, (o sea el segmento1), se busca dentro de la tabla de Mapa y Nombres de Segmentos lo solicitado, una vez hallado el dato se toma como información el apuntador a la Tabla de Estado de Segmentos

Activos, de la cual se obtiene la dirección del segmento buscado y sumándole el desplazamiento al que se desea acceder se obtiene la dirección real en la cual se encuentra el dato o instrucción deseados.

Podemos asimilar, en principio, este tipo de manejo, al de paginación, con la fuerte observación de que los segmentos son todos de longitud distinta, o sea tendrán un tamaño de acuerdo a una razón lógica, luego es muy importante conocer su dirección de comienzo y su longitud, elemento que se deberá utilizar para no permitir el acceso a direcciones no propias.

El campo Permiso, que generalmente será un apuntador a una tabla asociada, indicará las formas de acceso a ese segmento desde los distintos espacios de direcciones, identificados por sus BCP. Este campo está dado principalmente pues en este sistema se realiza con frecuencia la compartición de segmentos, para evitar la proliferación de un mismo código en función de las veces que es invocado. Obviamente estos códigos deberán ser reentrantes cuando sean compartidos.

Veamos cuál es el manejo de las tablas cuando se da la situación de que se invoca un segmentos que nunca antes habían sido invocado.

Supóngase que Sub2, invoca, por primera vez (para él), la subrutina SENO.

Si es la primera vez, significa que aún no tiene resuelta sus direcciones para ese segmento (Ver más adelante resolución de direcciones). Como aún no está resuelta, significa que se generará una interrupción para ese efecto.

La rutina que recibe esta solicitud, revisará la propia Tabla de Mapas y Nombres de Segmentos, con el parámetro SENO, al encontrarla, identificará ese segmento como el número 3 y devolverá esa información al segmento Sub2, con lo cual se realizará la vinculación para esta sesión.

En caso de haber solicitado el acceso a un segmento que no se encontrara en su propia Tabla de Mapas y Nombres de Segmentos, se debería buscar en la Tabla de Segmentos Activos, si allí se hubiese hallado, se consultaría el Permiso correspondiente, que en caso de ser válido provocaría que en la Tabla de Mapas y Nombres de Segmentos se generara una entrada para este nuevo segmento con el número correspondiente para este espacio de direcciones y se procedería a su posterior vinculación. Si el permiso hubiese sido no válido se adoptaría algún mecanismo de espera o cancelación.

Si el segmento tampoco hubiese sido hallado en la Tabla de Segmentos Activos, eso significa, que ninguno de los procesos actualmente en ejecución lo invocó, en consecuencia, es necesario generar una entrada en la

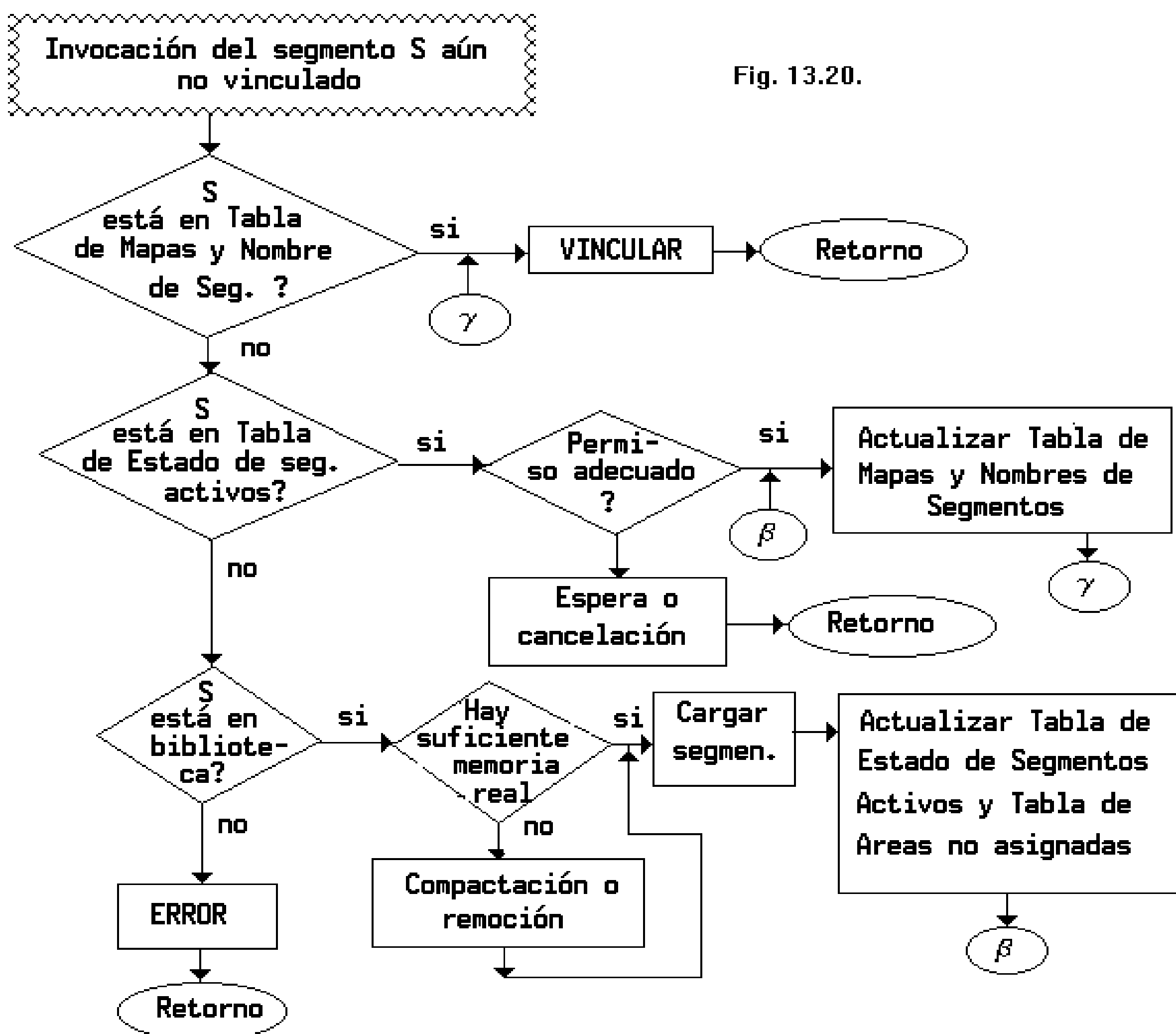


Fig. 13.20.

Tabla de Segmentos Activos, buscar el segmento en biblioteca, verificar la posibilidad de cargar el segmento en memoria consultando la Tabla de Areas no Asignadas, y si es posible cargarlo, completar los datos de la Tabla de Segmentos Activos, actualizar la Tabla de Mapas y Nombres de Segmentos del solicitante y vincular.

En caso de que no exista espacio suficiente en memoria real será necesario aplicar compactación o remoción de otros segmentos.

Luego, la actividad de búsqueda de segmentos aún no vinculados se puede diagramar como se ve en la Fig. 13.20.

13.11.2 - **Encadenamiento de Segmentos o Vinculación Dinámica**

Ya se mencionó que la vinculación se realiza durante la ejecución. Para poder cumplir esto se dice que el direccionamiento a otros segmentos se realizará de forma indirecta.

O sea que un llamado del tipo CALL SUB1 se transformará, por intervención del compilador en la siguiente pieza de código:

Fig. 13.21.

Donde:

*	indica direccionamiento indirecto.
1	registro con dirección de retorno
0/104	dirección de dirección

El bit en 1 en 104 indica dirección no resuelta

0/108	dirección del nombre del segmento
SUB1	nombre del segmento

Si durante toda la ejecución de este segmento no se ejecuta la instrucción que se encuentra en la dirección 050, SUB1 nunca será invocada, ni llamada, ni vinculada.

Si por el contrario se ejecuta el CALL, se irá a la dirección 104, donde el bit en 1 indicará dirección no resuelta y esto provocará la interrupción, Una vez resuelta ésta en la dirección 104 quedará:

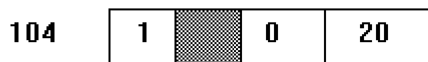


Fig. 13.22.

donde:

0	indica dirección resuelta
1	indica segmento 1
20	indica instrucción ejecutable (Entry Point)

Nota: el pasaje de parámetros es una convención, la cual puede ser Stack, registros, etc.

Hasta ahora hemos supuesto que las direcciones de los datos e instrucciones eran resueltos tomando la dirección del segmento y sumándole el desplazamiento, o sea que frente a una dirección S,d , se busca dentro de la Tabla de Segmentos el valor S, se toma su dirección y se le suma d, pero, qué sucede cuando un segmento se referencia a sí mismo, o cuando es compartido, siendo referenciado desde distintos espacios de direcciones con número de segmentos distintos ?

Obviamente no es posible cambiar en todas sus direcciones los números de segmento, y si además es compartido cada espacio de dirección lo ve como un número de segmento distinto, lo que obligaría a que dependiendo del llamador se cambiase el número de segmento.

Existen algunas propuestas, como por ejemplo, que todos los segmentos compartidos deberían tener un número de segmento tan alto, que ningún espacio de dirección pudiese llegar a ese número (Peterson 1985).

Esto tiene algunas desventajas, como ser que habría que conocer de antemano todos los números de segmentos, y si las entradas de las Tablas de Nombres y Mapas de Segmentos son correlativas, habría que guardar muchas entradas no utilizadas para llegar a esos números, con la consiguiente pérdida de espacio en memoria.

Otra propuesta es tener en claro cuando se direcciona dentro del segmento y cuando se direcciona a otro ó sea fuera del segmento. El Multics GE-645 implementó un indicador de dos estados (ON- OFF), estableciendo que si el indicador está en OFF se está direccionando dentro del segmento y la dirección base del segmento se encuentra en el Registro Base (Procesador) previamente cargada. Si está en ON, indica que la dirección se obtiene como S + d, si es búsqueda de dato, si se ejecutan instrucciones será necesario un cambio de Registro Base

(por ejemplo BIF 3/120 implica cambio de Registro Base). Nota: es mandatorio que los retornos siempre se realicen como retornos de subrutinas según la convención que se establezca y que esto implique la carga del Registro Base del Segmento llamador.

Una implementación sencilla es que cada segmento se ve a sí mismo como el segmento 0 (cero) (OFF). Nota: debe existir un registro de los posibles Registros Bases, que puede ser adicionado al BCP, o Tabla de Mapas y Nombres de Segmentos.

13.12 - **Administración de SEGMENTACION PAGINADA**

Esta administración junta las ventajas y desventajas de ambas administraciones. Comparte segmentos, aprovecha el uso de la paginación por demanda, con lo cual existen 2 tipos de interrupciones, falta de segmento y falta de página.

Debido al tamaño de las tablas es posible necesitar 3 accesos a memoria por cada acceso que se desee realizar, ya que es difícil que las tablas puedan ser contenidas en almacenamientos del procesador.

Su esquema puede verse en la Fig. 13.23.

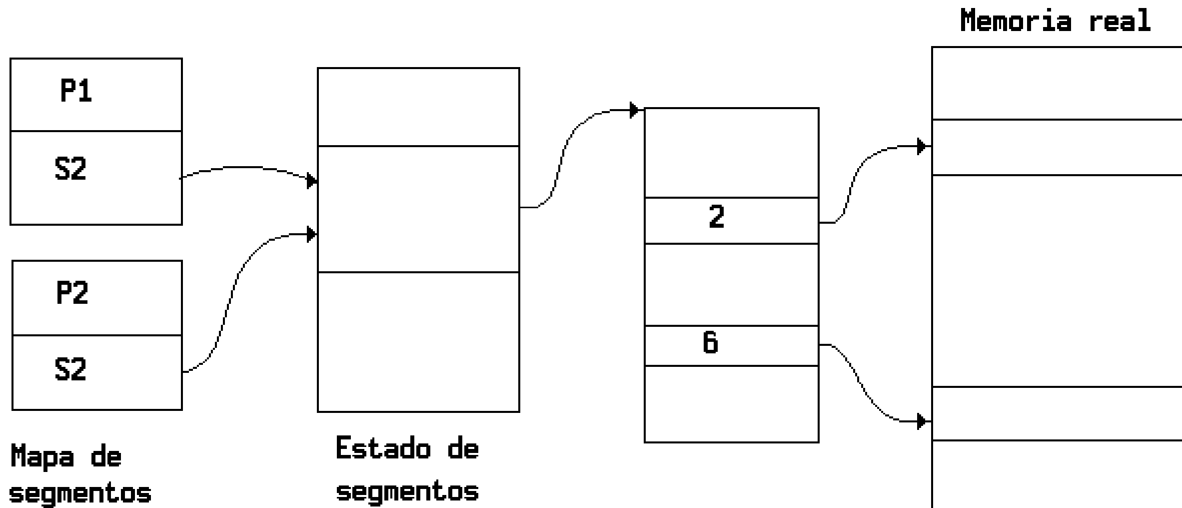


Fig. 13.23. - Administración de Segmentación Paginada.

Las necesidades de Hardware son:
- Protección
- Dirección Base
- Traducción de Direcciones
- Registro de Reubicación.

Las necesidades Software son :
- Tablas de Segmentos
- Dirección Base
- Longitud
- Presencia
- Cambio
- Uso
- Permisos
- Dirección Memoria Virtual
- Rutinas de atención por falta de Páginas
- Algoritmos de remoción
- Rutinas de Búsqueda de Páginas y Segmentos
- Rutinas de Vinculación en ejecución.

CAPITULO 14

ADMINISTRACION DE PERIFERICOS

14.1. - FUNCIONES

Los objetivos principales de una administración de periféricos son:
1) Llevar el estado de los dispositivos, lo cual requiere de mecanismos especiales. Estos mecanismos requieren de Bloques de Control de Unidades (UCB) asociados a cada dispositivo.
2) Determinar políticas para la asignación y desasignación de dispositivos. Esto es ver quién obtiene el dispositivo, por cuánto tiempo y cuándo. Por ejemplo una política de mucho uso de dispositivo intenta coordinar los pedidos de los procesos con la velocidad de los dispositivos de E/S.

14.2. - Tipos de Periféricos

En función de su asignación los periféricos pueden ser clasificados en:
i) Dedicados: Un dispositivo asignado a un sólo proceso.
ii) Compartidos: Un dispositivo compartido por varios procesos.
iii) Virtuales: Un dispositivo físico (generalmente de tipo dedicado) es simulado sobre otro dispositivo físico (de tipo compartido).
Recordemos que una vía de comunicación la podemos esquematizar como se ve en la Fig. 14.1.

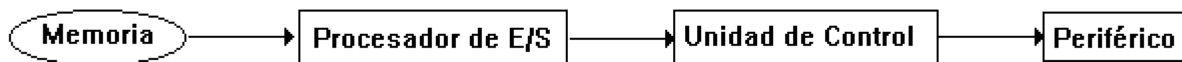


Fig. 14.1.

Un procesador de E/S controla la operación de E/S. Una unidad de control básicamente detecta errores de paridad en el envío de información y a veces es capaz de corregir esa información si tuviera una inteligencia suficiente. Además tiene buffers de sincronismo con los cuales compensa diferencias de velocidad entre los periféricos y el procesador.

En algunos sistemas se mantiene la estructura de la Fig. 14.2 que se suele denominar *String de Periféricos*. La cantidad de transferencias del string a memoria es una sola.

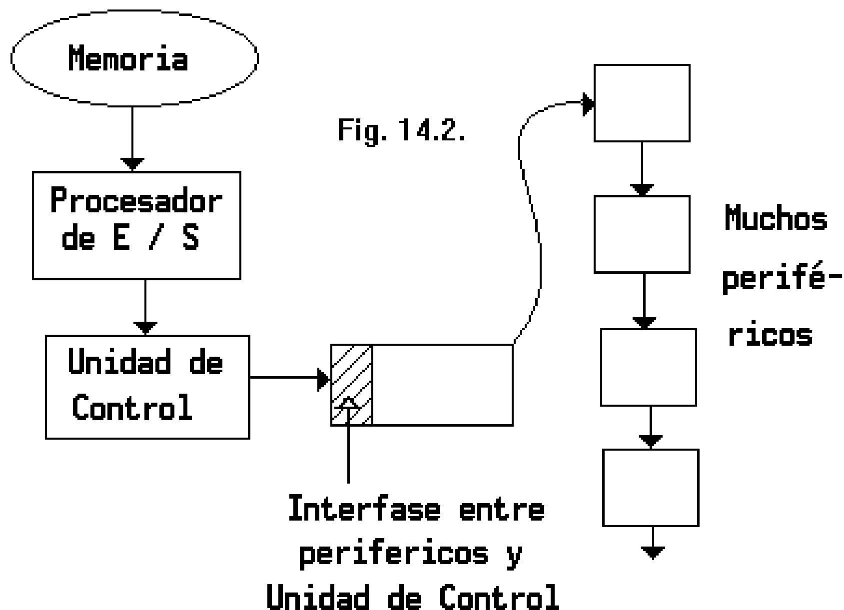


Fig. 14.2.

En todos los casos lo que se busca es:

a) confiabilidad en la transferencia y
b) velocidad.

Para ello se recurre a conectar los periféricos de distintas maneras como ser la que se ve en la Fig. 14.3.

Si deja de funcionar algún procesador de E/S todavía podremos acceder al periférico pasando por el otro. Esto se denomina comúnmente tener *Caminos Alternativos* para llegar a los periféricos.

Asimismo también es deseable proveer la posibilidad de transferencias simultáneas si el camino lo permite (Fig. 14.4).

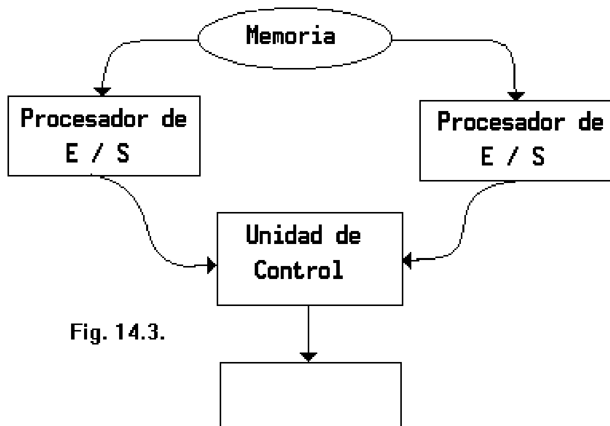


Fig. 14.3.

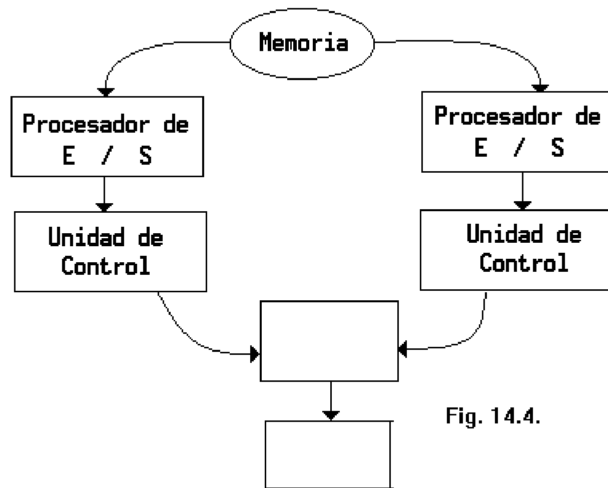
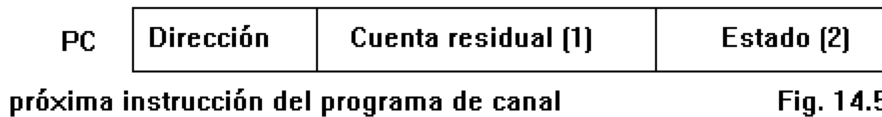


Fig. 14.4.

14.3. - **Canales y Unidades de Control**

Recordemos que un CANAL es un procesador especializado en operaciones de E/S. Las instrucciones que manejan son comandos de canal y sirven para dar órdenes al periférico y controlan la transferencia de la información. Al ser un procesador tiene su palabra de control, la cual puede esquematizarse como sigue (Fig. 14.5):



PC	Dirección	Cuenta residual (1)	Estado (2)

próxima instrucción del programa de canal

Fig. 14.5

(1) Indica cuántos bytes del último comando no han sido procesados. Usualmente su valor es 0 salvo una terminación de E/S anormal. En el comienzo contiene la longitud del buffer. A medida que se transfieren los bytes a memoria se decrementa. Cuando llega a 0 se produce una interrupción por fin de E/S.
(2) Indica si la operación se completó o si tuvo error.

Bajo un esquema de Procesador de E/S, Unidad de Control y Periférico, un periférico puede ser identificado por medio del camino a seguir por la información hasta llegar a él, por ejemplo Procesador de E/S 1, Unidad de Control 5, Periférico 3; luego ese periférico se identifica como "153".

Cuando un programa emite una instrucción de arranque (SIO 153 Start Input Output o IES de Inicialización de E/S) con Canal 1 - Unidad de Control 5 - Periférico 3, éste accede a una predeterminada dirección de memoria que tiene la dirección de la primera instrucción del programa del Procesador de E/S. El Procesador de E/S procesa estos comandos que tienen el formato de la Fig. 14.6.

Codop	Dirección del Buffer	Longitud (bytes)	Señales

Fig. 14.6.

codop: indica si son operaciones de transferencia, acciones al periférico sin transferencia, transferencia de control dentro del programa del procesador de E/S.
direcc.buffer: indica dónde se colocará o de dónde sacarán los datos de memoria.
long.bytes: indica la longitud.
señales: indican por ejemplo, si se concatenan las instrucciones del procesador de E/S, etc.

Con el campo de señales se logra la solución al problema del buffer a caballo entre 2 páginas (en administración de memoria paginada). Si se tiene un buffer cortado entre dos páginas, no se puede hacer otra cosa más que ejecutar dos instrucciones de procesador de E/S, una de las cuales tendrá la dirección de memoria inicial y la longitud hasta terminar la página, y concatenarla con otra instrucción que tendrá la nueva dirección y la longitud restante. Es decir el programa de transferencia contendrá instrucciones del siguiente tipo:

1ero) long.buffer = L1 + L2
2do) transferencia D1, L1
3ro) transferencia D2, L2
4to) señal fin

La interrupción de E/S se producirá al finalizar la ejecución de la segunda transferencia.

Cuando un procesador de E/S interrumpe al procesador debe informar si la operación de E/S finalizó correctamente, si hubo algún tipo de incidentes, etc.

Ejemplos de programas para Procesador de E/S son:

Para DISCO

Posicionamiento (cilindro, cabeza)

Búsqueda del bloque (sector)
Reintento
Transferencia (Dir, Long)
Para CINTA
Rebobinar
Saltear registro 1
Saltear registro 2
Escribir registro 3 de buffer 1
Escribir registro 4 de buffer 2
Escribir marca de cinta
Rebobinar y descargar

Dependiendo de la inteligencia de los elementos que componen la vía de comunicación (Unidad de control y periféricos), éstos pueden enviar órdenes y desconectarse (posicionamiento, rebobinar) aprovechando así tiempos "muertos".

Por otra parte las Unidades de Control tienen las siguientes funciones:

* Sincronizar la información (para ello cuenta con buffers de sincronismo para compensar diferencias de velocidad);
* En algunos casos serializa y des-serializa la información;
* Chequeo de paridad y detección de errores;
* Controlar movimiento del periférico (rebobinar, posicionamiento del brazo, etc.).

14.3.1. - **Tipos de Canales**

Selector: Desde que comienza a ejecutar un programa de canal para un periférico permanece conectado a él, aún cuando las operaciones no impliquen transferencia de información (Fig. 14.7).

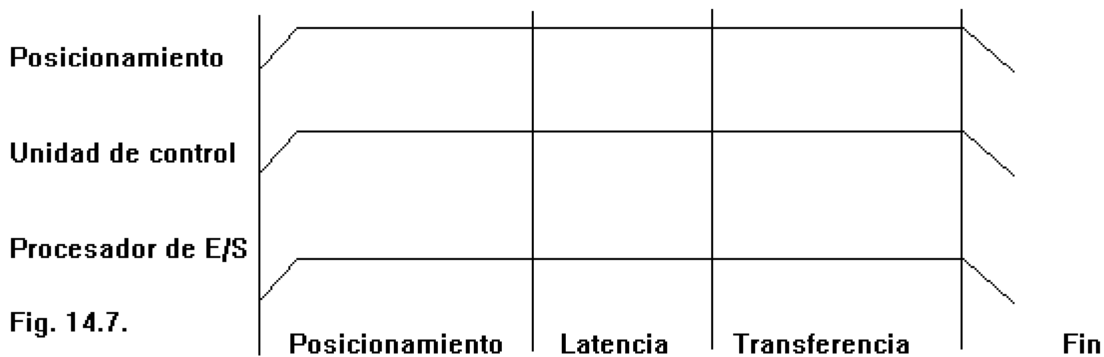


Fig. 14.7.

Solo sirve a un dispositivo a la vez. Normalmente se usa con dispositivos rápidos (cintas, discos y tambores), por lo tanto el pedido de E/S se completa rápido y se selecciona otro dispositivo.

Multiplexor: Se comparte entre varios programas de procesadores de E/S, o sea cuando da una orden de movimiento a un periférico, hasta que éste la complete, puede desconectarse del mismo y emitir otro comando del procesador de E/S para otro periférico. Debe estar conectado con el periférico siempre que exista transferencia de información. (Si el periférico es inteligente la unidad de control también puede desconectarse y atender a otro periférico).

Cuando el periférico está posicionado emite una señal con lo que vuelve a activar a la unidad de control y al procesador de E/S. Mientras dura la transferencia de información permanecen los tres activos (Fig. 14.8).

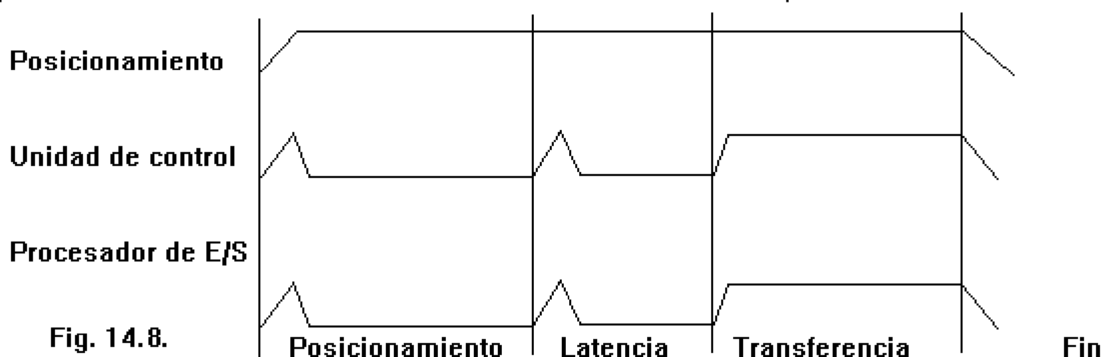


Fig. 14.8.

Puede llegar a ocurrir una desincronización: un periférico encuentra el sector, emite la señal pero el canal está ocupado con otro pedido. Entonces se pierde el sector y es necesario reintentar la búsqueda del mismo.

14.4. - **Tipos de Periféricos**

Existen distintas clasificaciones para periféricos pero lo que nos interesa es determinar si ese periférico debe ser de uso exclusivo para un proceso dadas sus características.

Por ejemplo una cinta no tiene sentido que se asigne a más de un proceso. Ese periférico va a estar dedicado solamente a uno.

Aquellos que pueden ser compartidos a lo largo de un lapso de tiempo, los llamaremos compartidos. El caso típico es el disco o el tambor magnético, que permiten acceso al azar. Si bien el periférico va a estar dedicado durante la transferencia no es necesario que lo esté a lo largo de todo el proceso.

La última clase que existe es la de periféricos virtuales: son periféricos que se los simula de alguna manera. Si tenemos periféricos dedicados, nos obligaría a ejecutar un solo proceso por vez que use esos periféricos. Si podemos simularlos, podríamos ejecutar más de un proceso de ese tipo simultáneamente. El ejemplo típico es el SPOOLing (Simultaneous Peripheral Operation On-line). En el apartado 14.11 trataremos más extensamente esta técnica.

14.5. - **Técnicas para la administración y asignación de periféricos**

Dependiendo del tipo de periférico que se trate existen diferentes técnicas de asignación del mismo. A saber:

1) dedicados: se asigna a un trabajo por el tiempo que dure el trabajo. Ej: lectora de tarjetas, cinta, impresora.

2) compartidos: pueden ser compartidos por varios procesos. La administración puede ser complicada. Se necesita una política para establecer que pedido se satisface primero basándose en :

a) lista de prioridades,

b) conseguir una mejor utilización de los recursos del sistema;

3) virtuales: Algunos dispositivos que deben ser dedicados (impresoras) pueden convertirse en compartidos a través de una técnica como el SPOOLing. Se hace una simulación de dispositivos dedicados en dispositivos compartidos. Por ejemplo un programa de SPOOLing puede imprimir todas las líneas de salida a un disco. Cuando un proceso escriba una línea, el programa de SPOOL intercepta el pedido y lo convierte a una escritura en disco. Dado que un disco es compartible, convertimos una impresora en varias impresoras "virtuales". En general se aplica esta técnica a periféricos dedicados lentos.

14.6. - **Políticas de asignación para periféricos Dedicados**

Para los periféricos de tipo dedicados se puede realizar su asignación al proceso en tres momentos diferentes :

* al comienzo del trabajo
* al comienzo de la etapa
* al realizarse la instrucción Open

Asignando al comienzo del trabajo estaremos seguros que cada vez que un proceso necesite el periférico, lo va a encontrar disponible. Pero si lo que tenemos es una larga secuencia de programas y sólo el programa que ejecuta en la última etapa es el que hace uso de ese periférico dedicado, hemos desperdiciado el periférico durante la ejecución de los programas anteriores.

Asignando al comienzo de la etapa intentamos entonces evitar la ociosidad de los periféricos y asignarlos sólo en el momento previo de las necesidades del programa. Puede ocurrir que en el momento de necesitar ejecutar esa etapa el periférico no se encuentra disponible y debamos tomar una decisión entre esperar a que el periférico se libere o cancelar el trabajo. A pesar de todo, todavía puede pasar un tiempo hasta que realmente se haga uso del dispositivo en esa etapa.

Asignarlo en el momento de la instrucción Open significa que ahora hacemos la asignación en el momento de querer usar realmente el dispositivo. Puede ocurrir, en este caso también, que no se encuentre disponible y habrá que tomar la decisión de esperar o cancelar. La desasignación generalmente se realiza en el momento de ejecutar un Close.

14.6.1. - **Asignación parcial y total de periféricos Dedicados**

Podemos hacer una asignación total o parcial del periférico.

En la asignación total el SO no entregará la totalidad de los periféricos dedicados que se necesitan hasta no tenerlos a todos disponibles. Si un proceso necesita 3 cintas, hasta que no haya 3 cintas libres no se le asignan, ni a nivel de etapa ni a nivel de trabajo.

En la asignación parcial a medida que el SO encuentra periféricos libres, los va asignando a los procesos, aunque no tenga la totalidad de los dispositivos que ese proceso requiere, esta política de asignación es la que se sigue cuando se asigna en el momento del Open.

Puede ocurrir en este caso un abrazo mortal (deadlock); ej. El proceso P1 pide 3 cintas, el proceso P2 pide 3 cintas y el sistema cuenta sólo con 4 cintas. Le asigna 2 a P1 y 2 a P2. P1 tiene 2 cintas y está esperando una más. Lo mismo ocurre con P2. Ninguno va a obtener el recurso que le falta pues lo tiene el otro. El caso de asignación parcial se debe controlar muy bien con el fin de evitar abrazos mortales, o evitar también el tener que matar algún proceso involucrado en un deadlock.

14.7. - **Políticas de asignación para periféricos Compartidos**

En este caso la asignación se hace en el momento de usarlos, es decir, en el momento de lanzarse la operación de E/S. Se arman colas de espera de disponibilidad de periféricos, unidades de control y procesadores de E/S. Para ello es necesario el uso de tablas. Estas tablas se relacionan formando una estructura de árbol.

14.8. - **Base de Datos para Administración de periféricos**

La base de datos para manejar una administración de periféricos consta de tres tablas cuyo contenido se explicita a continuación (Fig. 14.9, 14.10 y 14.11).

En el **Bloque de Control de Periférico** tendremos la siguiente información:

Identificador del dispositivo
Estado del dispositivo
Lista de unidades de control a las que está conectado el dispositivo
Lista de procesos esperando este dispositivo

En el **Bloque de Control del Procesador de E/S** tendremos la siguiente información:

Identificador del canal
Estado del canal
Lista de unidades de control a las que está conectado el canal
Lista de procesos esperando el canal

En el **Bloque de Control de la Unidad de Control** tendremos la siguiente información:

Identificador de la unidad de control
Estado de la unidad de control
Lista de dispositivos conectados a esta unidad de control
Lista de canales conectados a esta unidad de control
Lista de procesos esperando a esta unidad de control

14.9. - **Rutinas del Sistema**

Para estos dispositivos la asignación es dinámica, y participarán en este proceso los siguientes módulos:

Controlador de tráfico de E/S (Fig. 14.12)

- Lleva el estado de los dispositivos, unidades de control y procesadores de E/S por medio de bloques de

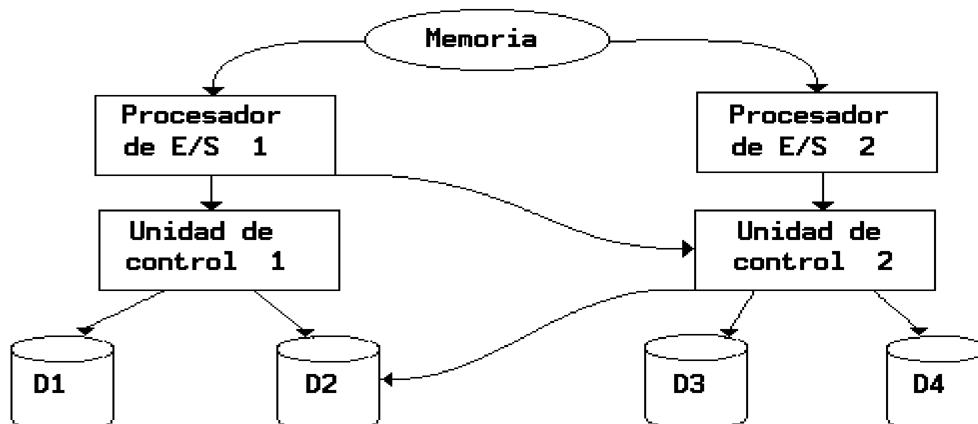


Fig 14.12.

control.

- Determina la posibilidad de una operación de E/S y la posibilidad de caminos alternativos

Cada bloque de control que maneja el controlador de tráfico de E/S servirá para poder determinar si es posible la operación y por qué camino. Cuando queremos hacer una operación de E/S, es necesario hallar toda la vía de comunicación. El proceso se encolará para que se le asigne un periférico, luego para que se le asigne una unidad de control y luego para que se le asigne un procesador de E/S. Sólo una vez que el proceso está a la cabeza de cada una de las tres colas, éste puede realizar su operación de E/S.

Planificador de E/S

- Decide qué proceso toma el recurso. Determina el orden en que se asignarán a los procesos los dispositivos, unidades de control y procesadores de E/S, ya sea por algoritmos o por alguna prioridad.

Una vez que hay un orden, es el controlador de tráfico de E/S el que determina que puede ser satisfecho el pedido. Las políticas a considerar son las mismas que usamos para procesos, salvo que una vez que un programa de procesador de E/S se lanza no se interrumpe sino hasta que termina.

Manipulador de periféricos

Existe uno por cada tipo de periférico que haya en el sistema.

- Arma el programa del procesador de E/S
- Emite la instrucción de arranque de E/S
- Procesa las interrupciones del dispositivo
- Maneja errores del dispositivo
- Realiza una planificación de acuerdo al periférico (estrategia de acceso).

14.9.1. – **Como interactúan las rutinas?**

Las rutinas mencionadas interactúan entre si para obtener que la E/S solicitada por un proceso se complete. Para ello es primero necesario obtener la ruta por la cual se realizará la transferencia de los datos desde o hacia el periférico hacia o desde la memoria central.

Las rutas se adquieren siempre comenzando desde el periférico y luego ascendiendo por la Unidad de Control y finalmente el canal. Esto obedece lógicamente a que la estructura jerárquica esta más poblada en las hojas (periféricos) que en su raíz (canal).

Una vez que la administración de la información ha determinado en qué lugar exacto del periférico se debe realizar la E/S, cuál es el periférico concretamente sobre el que se desea operar, le pasa esta información a la administración de periféricos.

La primera rutina que interviene es el manipulador de periféricos el cual con los datos que recibe construye el programa de canal ya que conoce cuál es el periférico concreto, de qué tipo de periférico se trata y sabe si la operación es de lectura o de grabación. Lo único que no conoce en este momento es qué ruta se utilizará para realizar la transferencia y, deja esta información aun sin completar.

Llama a continuación al Controlador de Tráfico de E/S para indicarle que este pedido va a esperar por el periférico en cuestión.

El Controlador de Tráfico de E/S coloca al pedido en el Bloque de Control del Periférico en la lista de procesos en espera del periférico.

A medida que el tiempo pasa y el periférico transfiere información de alguna E/S solicitada previamente (suponemos que el periférico esta ocupado) esta lista de procesos en espera del periférico se va poblando. Pero la lista no es de una capacidad infinita y el Controlador de Tráfico de E/S sabe cual es la cantidad de procesos que puede colocar en espera ya que es quién administra estas listas.

Una vez alcanzada una cota máxima de procesos en espera el Controlador de Tráfico invoca al Planificador de Procesos de E/S para que aplique su algoritmo y decida a cuál o cuáles de los procesos en espera se otorgará el uso del periférico cuando este se libere. Pueden llegar a ser más de un proceso ya que por características físicas del dispositivo puede convenir realizar dos o más E/S seguidas por cuestiones de optimización del uso del periférico.

Una vez obtenidos el/los ganadores de la selección hecha por el Planificador de Procesos de E/S éste invoca nuevamente al Controlador de Tráficos de E/S pidiéndole que ahora encole a estos procesos ganadores en la cola de procesos en espera de una Unidad de Control.

Hemos subido un escalón en la jerarquía de interconexión y ahora pasamos a competir para adquirir la Unidad de Control.

Nuevamente se repite el proceso mencionado anteriormente: se alcanza una cota, se invoca al Planificador de Procesos de E/S, se selecciona uno o más ganadores (aquí de la Unidad de Control) y se encola a los ganadores (por medio del Controlador de Tráfico de E/S) en la lista de procesos en espera por el Canal.

Una vez encolados en el Canal nuevamente repetimos el ciclo (cota, selección) y aquí ya hemos adquirido el escalón más alto en nuestra jerarquía de transferencia: el Canal y por ende toda la ruta Canal - Unidad de Control – Periférico.

Como ya obtuvimos la ruta entonces podemos invocar al Manipulador de Periféricos a efectos de que complete su programa de canal con la ruta obtenida. Se realiza finalmente (una vez que se libere el periférico) una última verificación de que la ruta está libre pero ahora en el sentido Canal → Unidad de Control → Periférico y si

estuviera libre entonces se lanza el comienzo de la E/S física, caso contrario habrá que esperar que se libere la porción de ruta que se encuentre ocupada.

Nótese que en ningún momento se optó por una ruta u otra aunque en realidad esta elección fue realizada por el Controlador de Tráfico de E/S ya que al encolar un pedido, su elección de hacerlo en una Unidad de Control u otra o un Canal u otro es la que implícitamente está determinando la ruta concreta que se utilizará finalmente para hacer la transferencia.

14.10. - **Algoritmos de planificación de E/S**

Bloqueo: es una técnica en la cual se agrupan muchos registros lógicos en un sólo registro físico. Esta técnica no sólo trae ventajas desde el punto de vista de nuestro programa, que va a estar menos tiempo demorado, sino que cada vez que haga una operación de READ va a tomar información que ya tiene en la memoria. Hay también un mejor aprovechamiento del periférico. Por ejemplo en una cinta entre registro y registro tengo un espacio entre registros desperdiciado (el IRG o Inter Record Gap) (Fig. 14.13).

Fig. 14.13.

Este espacio da el tiempo necesario para el arranque y la parada de la cinta. Existe un tiempo para frenar y otro para arrancar y adquirir la velocidad de transferencia que tiene el dispositivo. El IRG depende de la densidad de grabación. Para minimizar el Gap se utiliza la técnica de bloqueo.

Las ventajas de bloqueo son:

1) Menor cantidad de operaciones de E/S dado que cada operación de E/S lee o graba múltiples registros lógicos a la vez.
2) Menor espacio desperdiciado, dado que la longitud del registro físico es mayor que el IRG.
3) Menor cantidad de espacio en cinta cubierto cuando se leen varios registros.

Sus desventajas en cambio son:

1) Overhead de software y rutinas que se necesitan para hacer bloqueo, desbloqueo y mantenimiento de registros.
2) Desperdicio de espacio en buffers.
3) Mayor probabilidad de errores de cinta dado que se leen registros largos.

Buffering: Si tenemos un dispositivo o unidad de control con buffer podemos igualar velocidades con el procesador.

Por ej. Para dispositivos de muy baja velocidad se puede ir realizando una lectura adelantada del próximo registro a efectos de cuando el proceso lo requiera se transfieran varios registros desde el periférico a una mayor velocidad y se pueda liberar de esta forma el Canal de E/S. Se pueden tener también dos buffers y se va leyendo alternadamente de cada uno de ellos. El proceso no se vería detenido nunca, su pasaje a bloqueado en espera de E/S es prácticamente inexistente aunque no nulo (Fig. 14.14).

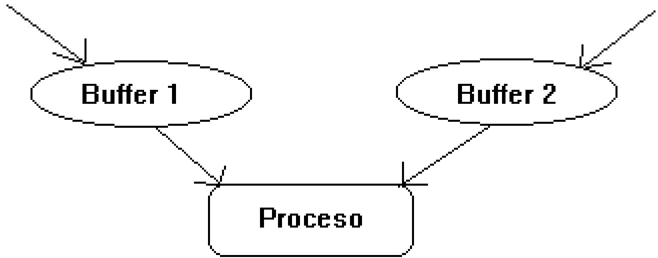


Fig. 14.14.

14.11. - **Algoritmos para encolar pedidos (en disco)**

El manipulador de periféricos puede utilizar técnicas de agrupación de pedidos sobre un mismo dispositivo para disminuir los tiempos mecánicos del mismo, algunas de ellas son :

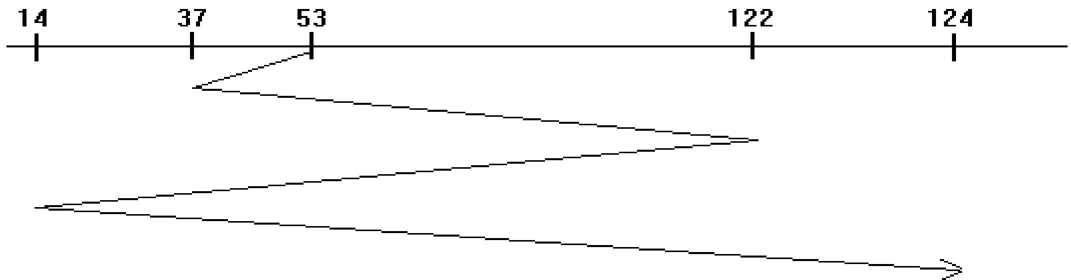


Fig. 14.15.

- **FIFO**: el primero que llega, es el primero en ser servido (FCFS - First Come First Served). Es el más simple. Sin embargo no da el mejor servicio promedio. Consideremos una cola que pide accesos a disco a las siguientes pistas: 37, 122, 14, 124 (estando inicialmente en la 53, Fig. 14.15).

Se recorren muchas pistas, podría haber sido más conveniente servir 122 y 124 juntos y luego la 14 y 37. El recorrido de la cabeza es de 319 pistas en total.

- **Más corto primero**: Se selecciona el pedido con el menor recorrido de búsqueda desde la posición actual de la cabeza, o sea, a la pista que está más cerca. El recorrido es de 149 pistas. Se debe conocer la posición actual de la cabeza (Fig. 14.16).

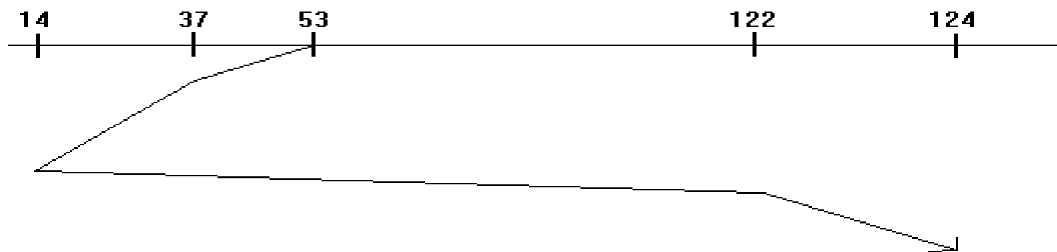


Fig. 14.16.

Pero esto puede ocasionar inanición (un pedido no es atendido nunca). Por lo tanto no es óptimo, aunque es mejor que el FIFO. Una solución a la inanición es encerrar a los pedidos en grupos.

- **Barrido Ascensor (SCAN)**: Aquí, la cabeza lectora/grabadora comienza en un extremo del disco y se va moviendo hacia el otro, atendiendo los pedidos a medida que llega a cada pista. Cuando llega al otro extremo vuelve. Para ello debemos conocer no sólo la posición de la cabeza sino también la dirección (Fig. 14.17).

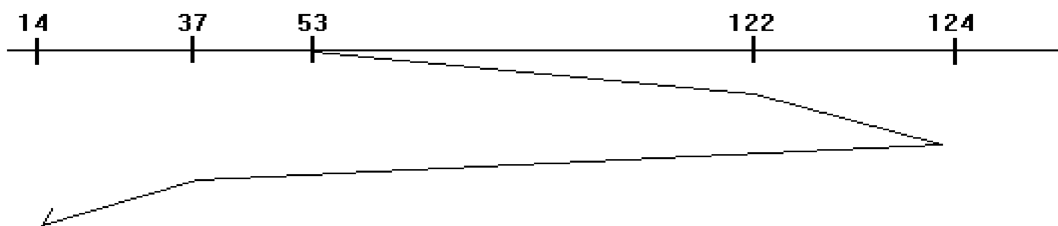


Fig. 14.17.

- **Barrido Circular (C-SCAN, circular SCAN)**: Está diseñado para atender los pedidos con un tiempo de espera uniforme. Mueve la cabeza lectora/grabadora de un extremo al otro del disco. Cuando llega al otro extremo, vuelve inmediatamente al comienzo del disco sin atender ningún pedido. Trata al disco como si fuese circular (Fig. 14.18).

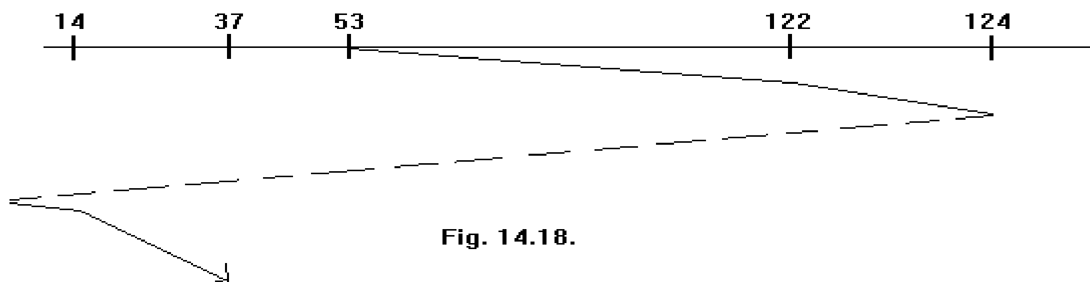


Fig. 14.18.

En la práctica ni SCAN ni C-SCAN mueven la cabeza de un extremo al otro. La cabeza se mueve hasta la posición del último pedido en esa dirección. Cuando no existen más pedidos en esa dirección se cambia la dirección del movimiento de la cabeza.

Estas versiones de SCAN y C-SCAN se llaman LOOK y C-LOOK.

Hay que tener en cuenta la ubicación de los archivos (directorios, archivos muy usados) en un dispositivo compartido. Conviene colocarlos en el medio del disco para reducir el movimiento de la cabeza. Si tengo archivos secuenciales, conviene que estén en pistas correspondientes a un mismo cilindro, de tal manera que no exista tiempo de posicionamiento para acceder a ese archivo.

- **Posición angular (sector queueing)** : Es un algoritmo para planificar dispositivos de cabezas fijas. Se divide cada una de las pistas en un número fijo de bloques, llamados sectores. El pedido ahora especifica pista y sector.

Se atiende un pedido por cada sector que pasa por debajo de la cabeza, aún cuando el pedido no está primero en la cola de espera. Cada sector tiene una cola.

Cuando un pedido para el sector i llega, se coloca en la cola del sector i. Cuando el sector i pase por debajo de la cabeza lectora/grabadora el primer pedido de la cola de ese sector es atendido.

Se puede aplicar también a discos de cabezas móviles, si hay más de un pedido en un mismo cilindro o pista. Cuando la cabeza se mueve a un cilindro en particular, todos los pedidos para ese cilindro pueden ser atendidos sin más movimiento de la cabeza.

14.12. - **DISPOSITIVOS VIRTUALES**

Dispositivos como las lectoras de tarjetas, perforadoras, impresoras, plotters, pantallas gráficas, presentan dos problemas que dificultan su utilización:

- Estos dispositivos trabajan bien cuando existe un flujo continuo y estable de pedidos a una tasa que se aproxime a sus características físicas (ej.: una tasa de impresión de 1000 líneas por minuto, etc.). Si un trabajo trata de generar pedidos más rápidamente que la tasa de performance del dispositivo, el trabajo debe esperar una cantidad significativa de tiempo. Por otro lado, si el trabajo genera pedidos a una tasa inferior, el dispositivo se mantiene ocioso gran parte del tiempo.
- Estos dispositivos se deben dedicar a un único trabajo a la vez.

Resumiendo, necesitamos una tasa de pedidos estable, mientras que los programas generan pedidos altamente irregulares (un programa con una tasa alta de cálculo puede imprimir muy pocas líneas, mientras que un programa con tasa alta de E/S puede imprimir miles de líneas en unos pocos segundos de CPU). La multiprogramación y el buffering reducen estos problemas, pero no lo hacen por completo.

14.12.1. - **Soluciones Históricas**

Estos problemas desaparecerían (o disminuirían substancialmente) si tuviéramos la posibilidad de utilizar algún dispositivo compartible (como ser un disco) para todas las entradas y salidas. Estos dispositivos pueden utilizarse simultáneamente para leer y/o escribir datos por varios trabajos, como mostramos en la Fig. 14.19.

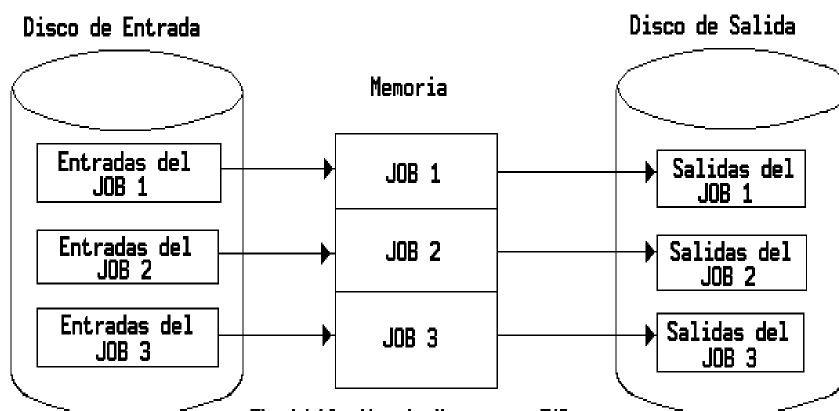


Fig. 14.19. - Uso de discos para E/S.

Además estos dispositivos tienen muy alta performance, especialmente si se hace bloqueo de datos, disminuyendo el tiempo de espera de los trabajos que requieren mucha E/S.

Pero esta especulación es imposible de llevar a la práctica, puesto que la utilización de dispositivos como las impresoras es necesaria para la mayoría de las aplicaciones.

14.12.2. - **Operaciones periféricas fuera de línea**

Una solución a este dilema puede encontrarse en el proceso de tres pasos ilustrado en la Fig. 14.20. En un primer paso, usamos una computadora separada cuya única función es leer tarjetas a máxima velocidad y almacenar la información correspondiente en un disco. Se pueden usar varias lectoras simultáneamente.

Como segundo paso, el disco conteniendo los datos de entrada ingresados en la computadora 1 se mueve a la computadora 2, encargada del procesamiento principal. Este paso es el correspondiente al mostrado anteriormente en la Fig. 14.19.

Puede haber muchos trabajos en ejecución, cada uno leyendo sus datos de entrada del disco; y escribiendo sus resultados en otro disco. Notar que en nuestro ejemplo es posible multiprogramar tres trabajos aunque existen sólo dos lectoras.

Finalmente, como tercer paso, se mueve el disco a una tercera computadora que lee del mismo los datos de salida, imprimiéndolos a alta velocidad e imprimiendo la información en las impresoras.

Este proceso se llama PROCESAMIENTO PERIFERICO FUERA DE LINEA, y las computadoras son conocidas con el nombre de COMPUTADORES PERIFERICOS, porque no hacen cálculo alguno sino que simplemente transfieren información de un dispositivo periférico a otro.

Podemos hacer varias observaciones a este esquema.

- Como las computadoras periféricas tienen tareas simples, pueden ser computadoras sencillas, lentas y baratas.
- Originalmente se utilizaban cintas en lugar de discos móviles.
- Aunque se solucionan los problemas planteados anteriormente, se introducen nuevos problemas, relacionados con:

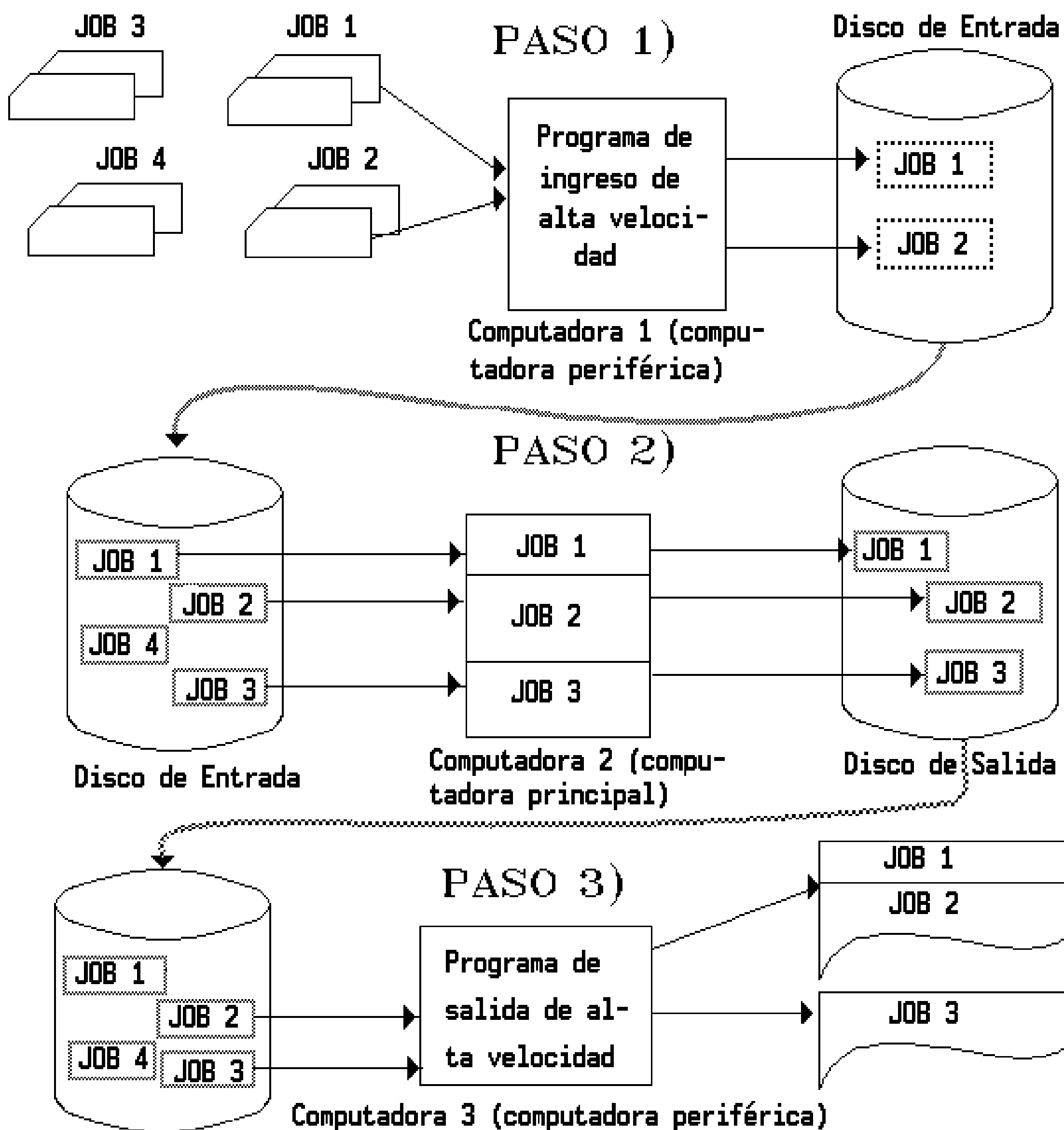


Fig. 14.20. - Operaciones periféricas off-line.

* Intervención humana: es necesario que alguien mueva los discos desde las computadoras periféricas hasta la computadora principal. Esto tiene como resultado la introducción de errores humanos y de ineficiencia.
* Tiempo de turnaround: como consecuencia inmediata de lo anterior, aumenta el tiempo entre la entrada y la salida de un trabajo del sistema. Otro motivo de aumento de este tiempo es la necesidad de que un disco se encuentre lleno antes de poder trasladarlo a la computadora principal.
* Es difícil proveer algún tipo de administración del procesador por prioridades. Sólo podemos usar PROCESAMIENTO BATCH.

14.12.3. - **SISTEMAS DIRECTAMENTE ACOPLADOS**

La mayor desventaja de la aproximación anterior era la necesidad de mover físicamente los discos entre el computador principal y las computadoras periféricos. Existen diversas soluciones a este problema. En la Fig. 14.21 mostramos una configuración en la cual los discos de entrada y salida están conectados físicamente al computador periférico y al computador principal.

Esta configuración se conoce como Sistema Directamente Acoplado. Normalmente se utiliza una única computadora periférica lo suficientemente poderosa, acoplada al computador principal.

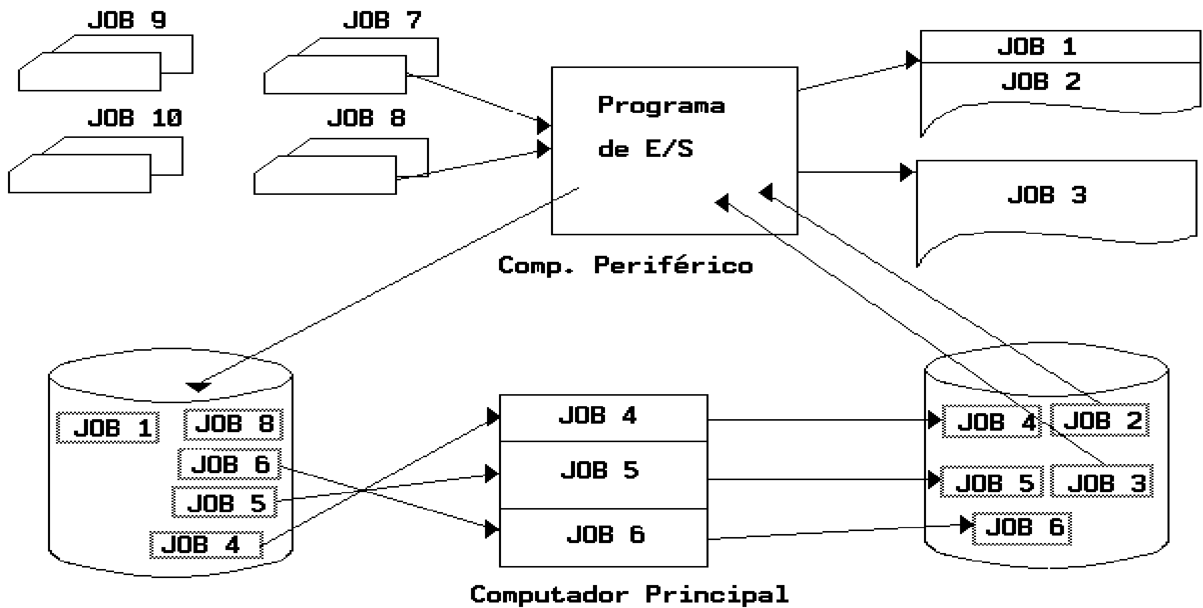


Fig. 14.21. - Configuración de un sistema directamente acoplado.

Esta aproximación elimina la mayoría de los problemas del procesamiento periférico fuera de línea. No es necesaria la intervención humana, y como los trabajos pueden procesarse tan pronto como la computadora periférica los coloca en el disco de entrada, no existe pérdida de tiempo de turnaround ni restricciones de administración del procesador.

Se puede utilizar un único disco como entrada y salida; pero el uso de un disco compartido de esta forma debe coordinarse cuidadosamente, porque si existen accesos frecuentes, puede haber conflictos que reduzcan la performance convirtiendo al disco un cuello de botella crítico.

14.12.4. - **PROCESADOR ADOSADO DE SOPORTE**

Otra variación de esta idea consiste de conectar directamente el periférico y la computadora principal vía una conexión de alta velocidad, como se ilustra en la Fig. 14.22.

En esta configuración, el computador periférico es un Procesador adosado de soporte.

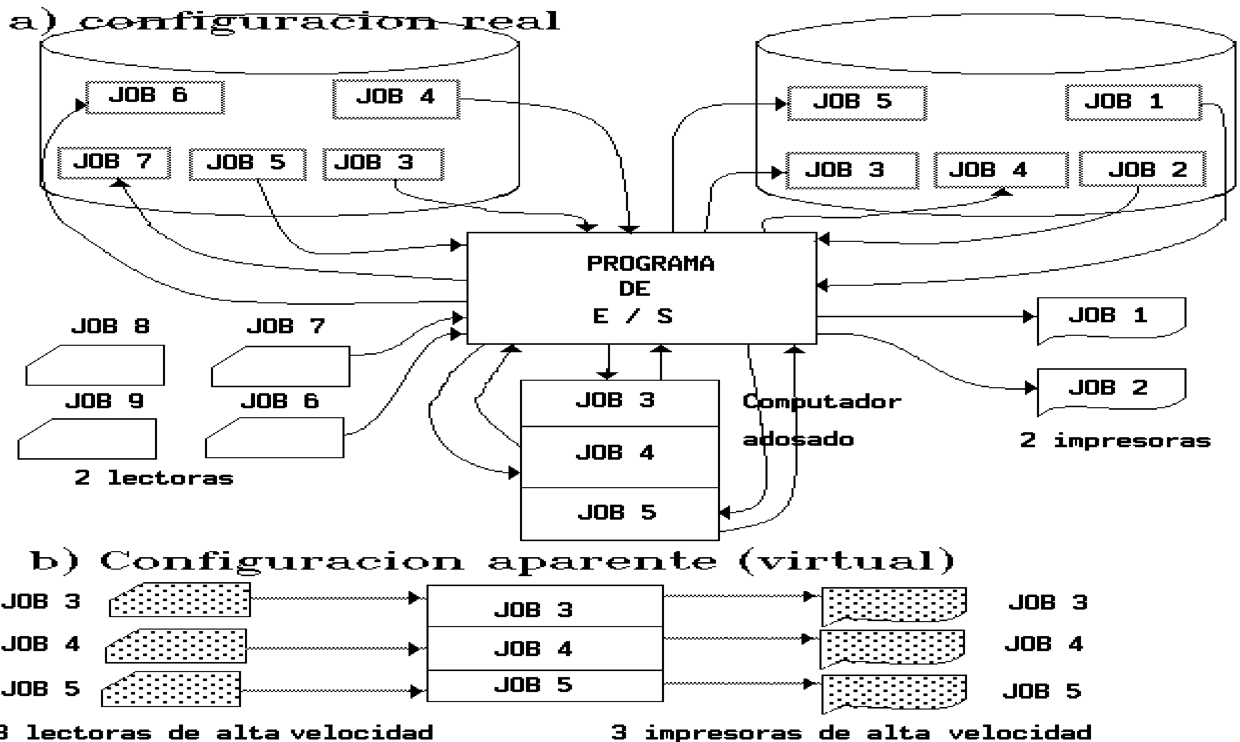


Fig. 14.22. - Configuración de Procesador Adosado de Soporte.

El procesador de soporte asume toda la responsabilidad de controlar los periféricos de E/S, así como los discos de E/S. También hace buffering y bloqueo, simplificando el trabajo del computador principal. En este modo, el procesador adosado tiene la apariencia de la existencia de múltiples lectoras e impresoras de muy alta velocidad (como vemos en la Fig. 14.22). Por analogía con el concepto de memoria virtual, el procesador adosado produce el efecto de dispositivos virtuales (es decir, muchos dispositivos y más rápidos que los que en realidad existen).

Las principales desventajas de este sistema son que son necesarios dos o más procesadores, y es posible que alguno de los procesadores esté ocioso o poco utilizado, mientras que el otro no puede manejar su carga. Este sistema, por lo tanto, no usa todos los recursos eficientemente.

14.12.5. - **SISTEMA VIRTUAL**

En todas las soluciones anteriores asumimos que existía una o más computadoras especiales, dedicadas al manejo de funciones de procesamiento de E/S. Considerando las facilidades de procesamiento disponibles en canales de E/S convencionales, así como las capacidades de multiprogramación, es necesario usar una computadora separada para manejar la E/S?.
En un sistema de SPOOLing, la computadora efectúa operaciones simultáneas de periféricos en línea (Simultaneous Peripheral Operations On-line). La idea se muestra en la Fig. 14.23.

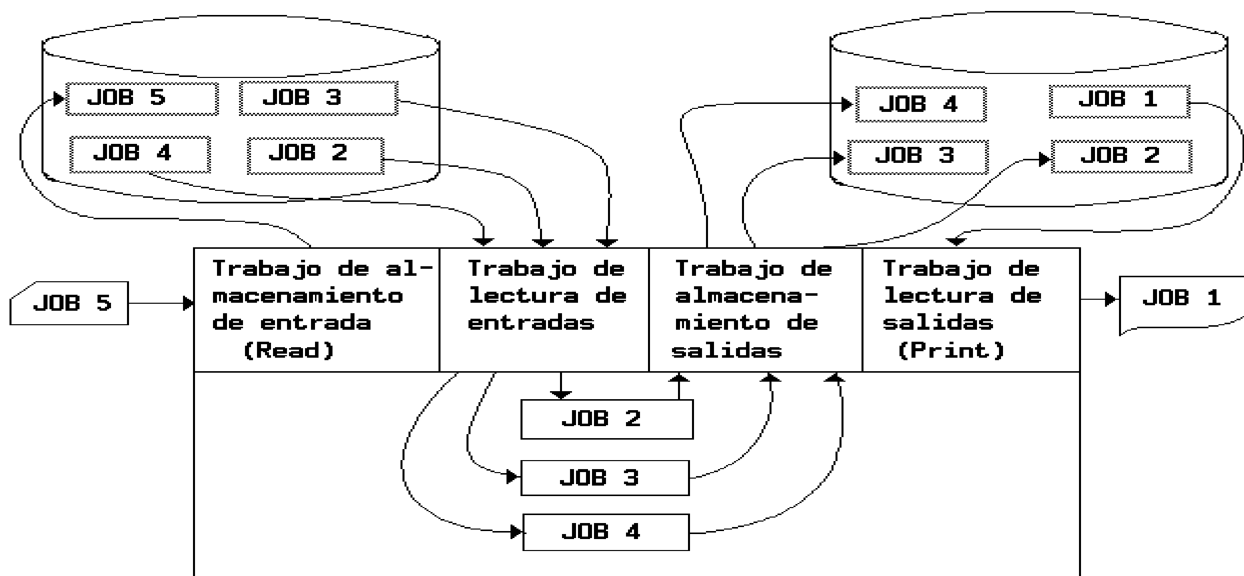


Fig. 14.23. - Sistema de Spooling.

En este sistema, determinados trabajos tienen las funciones del Procesador Adosado de Soporte, pero residen permanentemente en la computadora principal, y comparten el uso del procesador con trabajos normales vía multiprogramación. Muchas veces estos trabajos se incorporan al área de memoria del Sistema Operativo. Como debe haber coordinación y comunicación entre estos trabajos, se deben usar diversas facilidades de administración del procesador. Es necesario mantener información de los datos de entrada y salida que fueron almacenados en el disco. Finalmente, se debe lograr bloqueo, buffering y control de E/S, para lograr una buena performance.

La sencillez de implementar un mecanismo de Spool, depende directamente de las facilidades específicas provistas por la Administración de Memoria, Administración de Información, Administración del Procesador, y los componentes de Administración de Dispositivos del Sistema Operativo.

En un Sistema Operativo limitado, los programas de Spool pueden tener que hacer muchas funciones operativas; mientras que en un Sistema más complejo, el rol del Spool puede ser menor, dejándose el "trabajo duro" al Sistema Operativo. De hecho, los sistemas de Spooling mejoran tanto el sistema que se suelen proveer como parte de algunos sistemas operativos simples que no soportan multiprogramación.

14.12.5.1. - **DISEÑO DE UN SISTEMA DE SPOOLING**

Asumiremos que los programas de Spool son parte del Sistema Operativo, y que usan una Administración de Información propia y especializada. Estas son suposiciones típicas de sistemas operativos de pequeña y mediana escala. En estos sistemas hay dos BSV especiales:

READNEXT(BUFFER) : Lee la próxima tarjeta de entrada.
PRINTNEXT(BUFFER) : Imprime la próxima línea de salida.

En realidad, la tarjeta no es leída ni la línea impresa, ya que se están utilizando dispositivos virtuales; pero los trabajos que solicitan estos pedidos no lo saben.

Un sistema general de Spool de E/S puede subdividirse en cuatro componentes, como se ilustra en la Fig. 14.24. Estos componentes pueden agruparse de varias formas: por función (entrada o salida) o por el método de obtener el control (llamada al supervisor o interrupción).

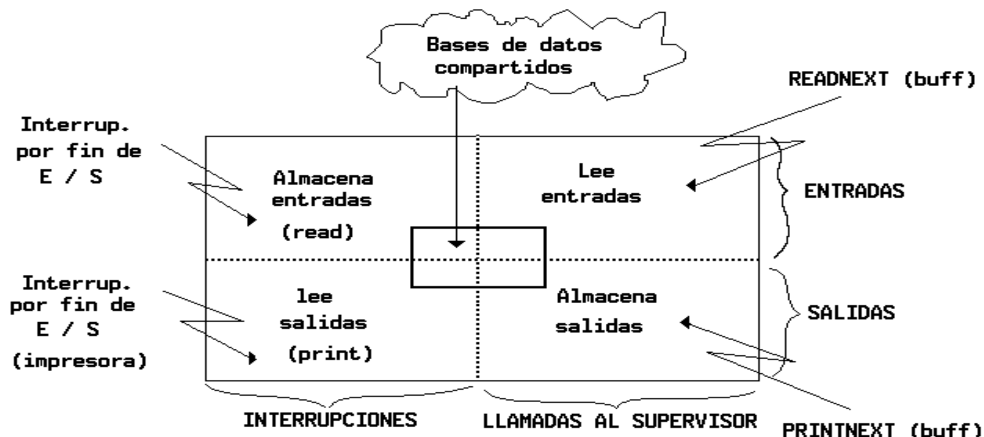


Fig. 14.24. - Estructura general del sistema de Spool.

14.12.5.2. - **SPOOL DE SALIDA**

Analicemos la función de Spool de salida (la del spool de entrada se maneja de una forma similar aunque esta técnica ha caído en desuso al ir desapareciendo la mayoría de los periféricos de entrada de baja velocidad).

El Spool debe efectuar dos operaciones principales:

- Interceptar la llamada a impresión por parte del proceso y almacenar físicamente cada línea impresa en un disco
- Recuperar cada línea impresa en el disco y transferirla a la impresora (impresión física).

Cuándo se efectúa realmente la operación de impresión? Debe hacerse independientemente del trabajo ya que, de hecho, *debe completarse luego de que finalice el mismo*.

Hay varias razones que justifican el hecho de que la impresión se realice luego de finalizado el trabajo o proceso que la generó:

- No es posible prever de antemano con qué frecuencia el proceso imprime. Si la impresión se comienza a realizar mientras el proceso aún está en ejecución puede ocurrir que se hayan impreso todas las líneas almacenadas en el spool y el proceso demora en generar más impresiones con lo cual la impresora está asignada (recordar que es un periférico dedicado) y sin embargo está ociosa.
- El contar con varios trabajos finalizados pendientes de impresión permite la planificación de los mismos para obtener el mejor tiempo de respuesta (por ejemplo: Mas corto primero –el de menor cantidad de líneas impresas-).
- Si un proceso genera una cantidad de líneas impresas excesiva es posible impedir que se imprima al observar que su spool en disco ha crecido desmesuradamente.

El sistema de Spool no es ni más ni menos un proceso más dentro del sistema con la característica que pertenece a las rutinas del sistema operativo y su función básica es interceptar los llamados a impresora por parte de los procesos y en lugar de realizar la impresión almacenar la línea impresa en un periférico compartido y luego de la finalización del proceso, transferir las líneas impresas almacenadas en el disco hacia la impresora.

Como cualquier proceso cuenta con su propia información, es decir, va llevando un registro de todos los procesos que inician impresión y mantiene los datos del lugar físico en el disco (que por extensión suele denominarse, disco de spool) en el que se encuentra almacenada la información impresa del proceso en cuestión.

Mantiene información sobre si el proceso aún está en uso de la impresora, o sea, su impresión aún no ha finalizado y sobre si el proceso ya finalizó su uso de la impresora y en consecuencia debe transferir la impresión al dispositivo físico, la impresora.

En un sistema que cuenta con spool para un proceso que pasa al estado de Terminado se agrega ahora un procesamiento especial por parte del sistema operativo que será la liberación del disco de spool (si el proceso generó impresión). Por lo tanto la rutina de spool interviene en las transiciones o estados de :

- Ejecutando a Bloqueado, interceptando el pedido de impresión ya que ahora se realizará una E/S al disco de spool en lugar de la impresora.
- Terminado, para transferir las líneas impresas desde el disco de spool hacia la impresora.

En la transición de Bloqueado a Listo, cuando finalice la E/S del disco de spool la rutina que atiende esa interrupción colocará al proceso nuevamente en estado de Listo

14.12.5.2.1. – **Un ejemplo**

Veamos con detenimiento un ejemplo concreto con un gráfico de tiempos.

Supongamos que contamos con un proceso que realiza durante toda su ejecución solamente dos impresiones y que además es el único proceso presente en el sistema.

Estado Proceso	E	B	B	E	E	B	B	E	E	E	T	T	T	T	T	T
CPU proceso	X			X	X			X	X	X						
CPU Spool		X				X					X			X		
Disco spool			Graba				Graba					Lee			Lee	
Impresora													X			X
Tiempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

En nuestro gráfico hemos despreciado el tiempo de las rutinas del sistema operativo excepto los de la rutina de spool (por ejemplo, la rutina de atención de interrupciones por fin de E/S, el planificador de procesos, etc.).

Veamos que ocurre en cada momento:

Tiempo 1: El proceso comienza a ejecutar y lanza una E/S sobre la impresora
Tiempo 2: La rutina de spool intercepta el pedido, genera información en sus tablas internas para este proceso, busca espacio en el disco para almacenar la línea impresa y finalmente lanza la E/S sobre el disco de spool.
Tiempo 3: Se comienza a realizar la E/S sobre el disco de spool para almacenar la línea impresa
Tiempo 4: El proceso retoma su ejecución una vez finalizada la E/S
Tiempos 5 a 7: se repite el proceso de generación de la línea impresa en el disco de spool
Tiempos 8 a 10: El proceso continúa su ejecución y finaliza en el tiempo 10
Tiempo 11: La rutina de spool luego de finalizado el proceso recupera la información del mismo del disco de spool para enviar la impresión. En este caso se recupera la primer línea impresa y se despacha a la impresora (de estar libre)
Tiempo 12: Se lee el disco de spool para obtener la primer línea impresa del proceso.
Tiempo 13: La impresora realiza la impresión física de la primer línea del proceso
Tiempo 14: La rutina de spool recupera la segunda línea del proceso
Tiempo 15: Se lee el disco de spool
Tiempo 16: Se imprime la segunda línea del proceso.

Como nuestro proceso es el único en el sistema no tenemos interferencia por parte de otros procesos, es decir, la impresora está libre y pudo asignarse ni bien terminó el proceso para imprimir sus líneas. Además ni bien finalizó la E/S del disco de spool nuestro proceso pudo recomenzar su ejecución y no debió esperar que otro proceso abandone el control de la CPU.

El gráfico no está hecho a una escala correcta. Lo correcto sería que el tiempo de la rutina de spool fuera muy pequeño (aunque no nulo) ya que el spool como rutina del sistema operativo está optimizada para que su ejecución tenga un bajo impacto en la performance del sistema.

Además el tiempo de la E/S de impresión debería ser mucho mayor comparado con el tiempo de la E/S del disco de spool.

14.12.5.2.2. – **Multiprogramación con y sin spool y Monoprogramación**

Nótese que comparativamente respecto de un sistema que no cuenta con spool ahora se agregan a los tiempos de cualquier proceso el tiempo de ejecución de la rutina de spool que es pequeño, pero no nulo y los tiempos del disco de spool que son tiempos bastante importantes por ser tiempos de E/S físicas a un disco.

En conclusión si se compara el tiempo de ejecución de un solo proceso, solo en el sistema, respecto del mismo proceso en un sistema sin spool y en un sistema de monoprogramación puede establecerse que el tiempo total del proceso, entendiéndose por tal desde que ingresa al sistema hasta que finaliza todo su procesamiento (impresión incluida) será:

Multiprogramación con spool >> Multiprogramación sin spool > Monoprogramación

En la multiprogramación con spool se agregan los tiempos de la rutina de spool más los tiempos del disco de spool que deben computarse dos veces (grabar el disco y luego leerlo para mandar la impresión) mientras que en la multiprogramación sin spool respecto de monoprogramación solo se agrega, principalmente, la rutina del planificador de procesos (eventualmente las rutinas de atención de interrupción por reloj en los esquemas de administración del procesador que poseen reloj de intervalos).

14.12.5. – **Beneficios del Spool**

Luego de lo antedicho en el párrafo anterior uno podría preguntarse, entonces cuál es el beneficio concreto de agregar Spool en un sistema?

Hay varios beneficios, a saber:

1) Mediante la técnica de spool es posible simular más periféricos de los que existen en la instalación. No sería posible ejecutar un proceso que utiliza dos impresoras simultáneamente para emitir dos listados si la instalación tiene una sola impresora y no tiene spool.

2) Mediante la técnica de spool es posible simular periféricos que no existen en la instalación. Si la impresora se rompe es posible ejecutar igualmente los procesos y luego llevarse el disco de spool a otra instalación y hacer las impresiones.
3) Los procesos, en un sistema que cuenta con spool, están menos tiempo en el circuito de estados Ejecución-Listo-Bloqueado. Como las impresiones se realizan en realidad sobre un disco el tiempo que el proceso permanece en estado Bloqueado es menor y por lo tanto vuelve pronto al estado Listo. Esto permite un mayor grado de multiprogramación en la instalación.
4) Los periféricos manejados por el spool, la impresora en nuestro caso, pueden funcionar a su máxima velocidad lo que redunda en un mejor aprovechamiento del mismo. Nótese que como el spool en el momento de la impresión física solamente está realizando una transferencia del disco a la impresora no existe ningún procesamiento de los datos y por ende la transferencia se puede realizar a la máxima velocidad soportada por el periférico.
5) La planificación de los trabajos de impresión que se realiza una vez finalizados los procesos redunda también en un mejor aprovechamiento del uso de las impresoras.

Hay que tener presente que el beneficio de la implementación del spool es un beneficio *para el sistema en su conjunto* y no para un proceso en particular.

14.13. – **TECNOLOGIA RAID**

La tecnología de discos RAID (Redundant Array of Independent Disks) fue originalmente concebida en 1987 por ingenieros en la Universidad de California en Berkeley. Ellos definieron originalmente los niveles 0 a 5 de Raid y luego se agregaron más niveles.

La idea subyacente en esta tecnología es proveer a través de hardware un mecanismo de tolerancia a fallas directamente implementado sobre un conjunto de discos en donde se almacenan los datos. La lógica de funcionamiento está implementada íntegramente por hardware

Esto suele formar parte de lo que se suele denominar "**almacenamiento estable**", ya que el mismo es tolerante a fallas.

Hay que indicar que el nivel 6 no es mejor que el nivel 0 y los niveles solo se utilizan para indicar la forma en que la información se almacena. La diferencia es, por supuesto, el costo el cual deberá evaluarse según la criticidad de los datos que se deseen almacenar.

14.13.1. – **Niveles de RAID**

Nivel 0

La información se reparte entre diferentes ejes de discos que son considerados en su conjunto como una sola unidad. Es lineal y no se mantiene ninguna información de paridad.

Nivel 1

Este es uno de los más conocidos ya que se trata de un mirror (espejo) de discos. Un disco es copia fiel del contenido de otro disco, o sea, se duplica el espacio en disco siendo uno de los discos backup del otro.

Nivel 2

En este nivel la información se reparte entre diversos discos a nivel de bit y cuenta con información de paridad.

Nivel 3

En este nivel la información se divide a nivel de byte y un disco del conjunto esta dedicado a almacenar la información de paridad (códigos de Hamming).

Nivel 4

Es similar a RAID 3, pero la división de la información se realiza en grandes pedazos.

Nivel 5

La información se reparte en bloques secuenciales a través de los discos junto con la información de paridad.

Nivel 6

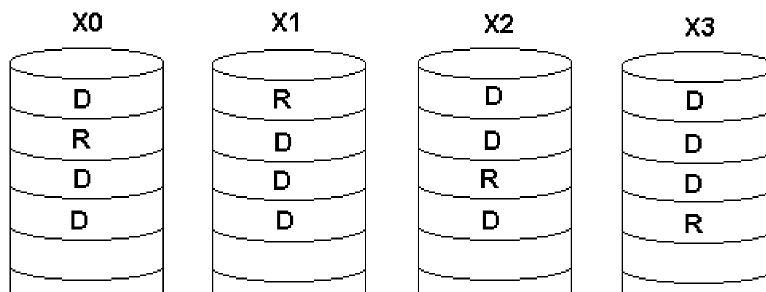
Es igual que RAID 5, pero agrega controladores de discos redundantes, buses extra, etc.

Nivel 7

En este nivel los buses de datos de E/S son asincrónicos. La jerarquía de dispositivos y buses de datos también es asincrónica.
Incluye un sistema operativo incorporado, orientado a los procesos de E/S, para manejar todas las transferencias de E/S a través de los discos.

14.13.2. – **Forma de almacenamiento de los datos**

Tomaremos como ejemplo un sistema con 4 discos implementados con tecnología RAID.



El dato original está compuesto por X0, X1 y X2 partes. En tanto que X3 se construye con una operación que resulta de las partes del dato original.
Digamos por ejemplo, que la operación aplicada es un XOR la que denotaremos con el símbolo $\oplus$.
Luego, se tiene :

$$X3 = X0 \oplus X1 \oplus X2$$

Supongamos ahora que el dato cambia en su parte de X1, luego :

$$X3' = X0 \oplus X1' \oplus X2$$

Ahora bien, como se sabe $A \oplus A = 0$, entonces podemos agregar lo siguiente:

$$X3' = X0 \oplus X1' \oplus X2 \oplus X1 \oplus X1$$

Pero $X0 \oplus X1 \oplus X2$ es igual a X3 con lo que queda

$$X3' = X3 \oplus X1' \oplus X1$$

Y esto ahorra el tener que leer todos los Xi de los otros discos.

Supongamos ahora que falla el disco que contiene X1, entonces se agrega a ambos miembros de la igualdad :

$$X3 \oplus \mathit{X3} \oplus \mathit{X1} = X0 \oplus X1 \oplus X2 \oplus \mathit{X3} \oplus \mathit{X1}$$

Y eliminando los miembros nulos se obtiene:

$$X1 = X0 \oplus X2 \oplus X3$$

Con lo cual el dato original de X1 se recupera con los otros datos almacenados en los otros ejes.

CAPITULO 15

ADMINISTRACION DE LA INFORMACION

15.1 - INTRODUCCION

Ya hemos visto con anterioridad el ciclo por el cual pasa todo trabajo desde que es submitido a un sistema de cómputo, hasta que eventualmente termina toda actividad asociada a este.

Ahora bien, qué implica poder llevar a cabo todos los objetivos presentes en un trabajo o tarea?. Obviamente, realizar todos los procesos pertenecientes al Trabajo en cuestión, para lo cual es necesario en el momento en que uno de dichos procesos se encuentra en estado de ejecutando tener disponibilidad de acceso a datos y/o programas que constituyen parte del proceso.

Es claro que no necesariamente toda esta información se encontrará en memoria al mismo tiempo, y tampoco necesariamente en el momento en que es reclamada por el procesador. Además, qué pasa con todo aquel conjunto de información que constituye el resto de trabajos que se encuentra en ese preciso momento en el sistema de cómputo?. Toda esta información se encuentra en lo que se da en llamar memoria secundaria o almacenamiento secundario (discos, cintas, etc.).

Por todo lo anteriormente expuesto, es claro que el sistema operativo debe constar de un módulo que se encargue del manejo de aspectos tales como el almacenamiento y recuperación de la información que se encuentra dentro del sistema. Resumiendo, podemos considerar que el Administrador de Información posee las siguientes funciones básicas:

1. Llevar rastro de toda la información en el sistema a través de varias tablas, siendo la mayor de ellas el Directorio de Archivos.

   Estas tablas contienen para cada archivo:
   - el nombre,
   - ubicación,
   - longitud (cantidad de registros del archivo),
   - longitud del registro lógico,
   - longitud del registro físico,
   - formato de registros (fijo, variable, etc.),
   - organización (secuencial, indexada, al azar, etc.),
   - fecha de creación,
   - fecha de expiración,
   - derechos de acceso,
   - extensiones.

   Las extensiones de un archivo obedecen al hecho de que a menudo no es posible almacenar el mismo en un fragmento contiguo del periférico. Por lo tanto en el campo de extensiones se indican la cantidad de fragmentos que existen de ese archivo y se inicia a partir de la primera extensión (usualmente denominada alocación primaria del archivo) una cadena que apunta a las otras entradas del Directorio en donde continúan los otros pedazos del archivo. En adecuación a esto el tamaño total del archivo es el de su alocación primaria más el tamaño de todas sus extensiones.
2. Decidir que política es utilizada para determinar dónde y cómo la información es almacenada. Algunos factores que influyen en esta política son:
   - utilización efectiva del almacenamiento secundario,
   - acceso eficiente,
   - flexibilidad a usuarios, y por último
   - protección de derechos de acceso sobre información pedida.
3. Asignación del recurso de la información. Una vez que la decisión de permitir a un proceso tener acceso a cierta información es efectuada, los módulos de ubicación de la información deben encontrar dicha información, hacer accesible dicha información al proceso y por último establecer los derechos de acceso apropiados.
4. Desasignación del recurso de la información. Una vez que la información no es necesitada más, las entradas en tablas asociadas a esta información pueden ser eliminadas. Si el usuario ha actualizado la información, la copia original de la información puede ser actualizada para posible uso de otros procesos.

Comúnmente al conjunto de módulos del Administrador de Información se lo referencia como el *Sistema de Archivos*. Es una meta al concebir un Sistema de Archivos que el mismo libere al usuario de problemas tales como la ubicación de la información que está referenciando, así como el formato real de dicha información en el sistema y el problema de acceso de E/S. Al quedar liberado el usuario-programador de tales escollos, puede concentrarse totalmente en su problema específico (estructura lógica y operaciones necesarias para procesar dicha estructura).

15.2 - MODELO GENERAL DE UN SISTEMA DE ARCHIVOS

Para el Sistema de Archivos que tomaremos como ejemplo en este capítulo, asumiremos siempre las posibilidades más sencillas. No obstante, a medida que se vayan explicando los distintos niveles de la implementación, se dejará constancia de sus deficiencias y posibilidades de mejora.

Consideraremos que todos los módulos de la implementación se encuentran dispuestos en niveles o capas, de tal forma que dado un módulo, este sólo depende de aquellos módulos que se encuentran en niveles inferiores a él y sólo efectúa llamadas hacia estos módulos. Son obvio las ventajas en cuanto a modularidad se refiere el concebir el Sistema de Archivos como una estructura jerárquica de este tipo. En la figura 15.1 podemos ver dicho modelo.

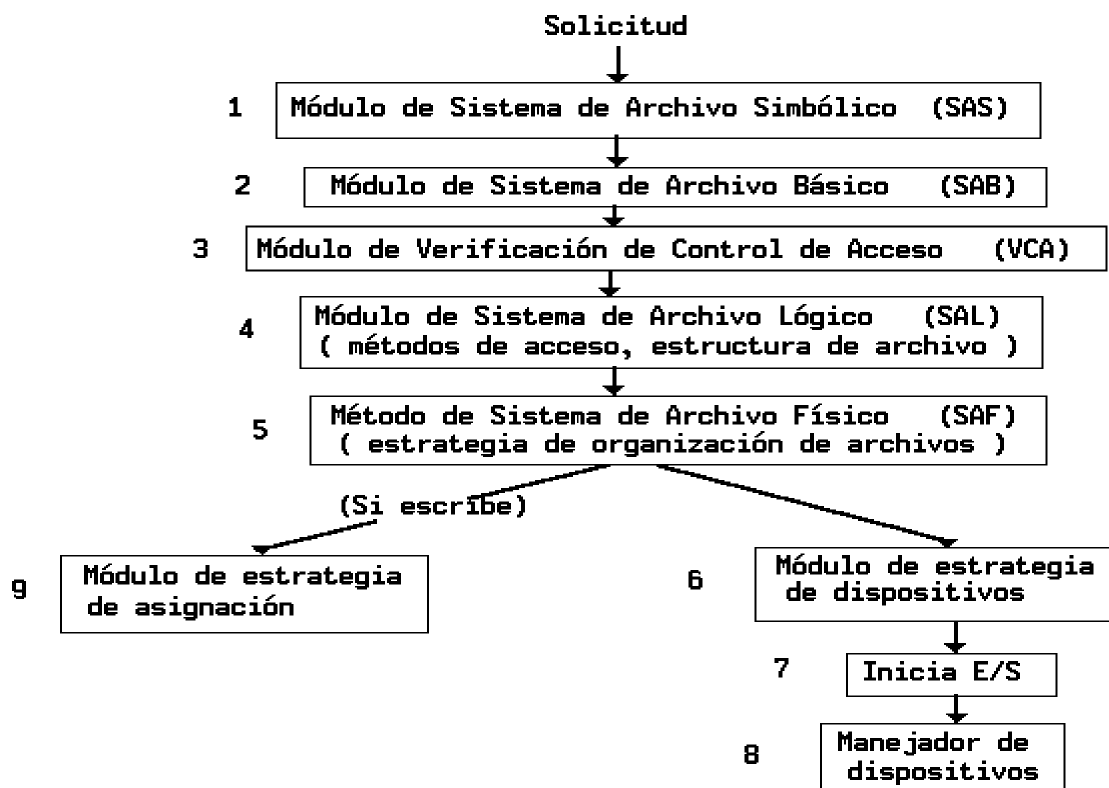


Fig. 15.1. - Modelo jerárquico de un sistema de archivos.

También es necesario destacar que aunque los detalles específicos dados a continuación pueden variar significativamente de un sistema a otro, la estructura básica es común a la mayoría de los sistemas actuales. Es decir, dependiendo del sistema específico que tratemos, algunos de estos módulos se fusionan, se desglosan en aún más módulos, o incluso algunos desaparecen. Aún así la estructura subyacente sigue siendo la misma.

15.2.1 - **ESTRUCTURA Y MANTENIMIENTO DEL DIRECTORIO DE ARCHIVOS**

Antes de describir el funcionamiento de los distintos módulos que conforman el Sistema de Archivos es necesario sentar algunas bases acerca de los siguientes ítems:

Estructura del Directorio de Archivos:

Nosotros visualizaremos al Directorio de Archivos como una tabla donde cada entrada en ella corresponde a un archivo. Definiendo a un archivo como una colección de unidades de información relacionadas llamadas registros.

Así, el contenido de una entrada del Directorio de Archivos de nuestro ejemplo contendrá los siguientes datos:

- Número de entrada,
- Longitud de registro lógico,
- Cantidad de registros lógicos,
- Longitud del registro físico,
- Formato de los registros,
- Organización,
- Dirección al primer bloque físico,
- Protección y control de acceso.

El Número de entrada en la tabla: se mantiene dicho número pues este se convertirá en el identificador del archivo dentro del sistema.

Desde el 2do al 7mo ítem de la tabla sirve para poder ubicar y mapear un registro lógico en su verdadera dirección física en el almacenamiento secundario.

La Protección y control de acceso indica qué usuario tiene derecho de acceso sobre el archivo y que tipo de operaciones se pueden realizar sobre el archivo (solo lectura, lectura/escritura, etc.).

Creación de una entrada del Directorio de Archivos:

Los comandos CREAR y BORRAR incorporan o eliminan respectivamente una entrada en el Directorio de Archivos. Así, una posible sintaxis del comando CREAR sería:

CREAR nombre-archivo, longitud-registro-lógico, cantidad-registros-lógicos, [ubicación-primer-bloque-físico], [formato], [organización]

Ubicación del Directorio de Archivos:

Si todo el Directorio de Archivos fuera mantenido todo el tiempo en memoria principal, se necesitaría una gran cantidad de memoria sólo para este propósito, por lo que surge la idea de mantener al Directorio de Archivos como un archivo dentro de un volumen de almacenamiento (cinta, diskette, disco, etc.).

No obstante se soluciona el problema del desperdicio de memoria principal, ahora nos encontramos ante el inconveniente de que la búsqueda de una entrada en dicho directorio puede ser considerablemente larga, si este constara, por ejemplo, de unas cuantas miles de entradas.

Este problema se soluciona si mantenemos en memoria aquellas entradas del Directorio que pertenecen a archivos que fueron referenciados anteriormente. En llamadas subsiguientes no habrá desperdicio de tiempo de E/S.

Muchos sistemas operativos constan de pedidos especiales tales como ABRIR y CERRAR (Open y Close respectivamente) un archivo determinado. ABRIR coloca en memoria la entrada en la tabla perteneciente al archivo en cuestión y CERRAR constituye su contrapartida.

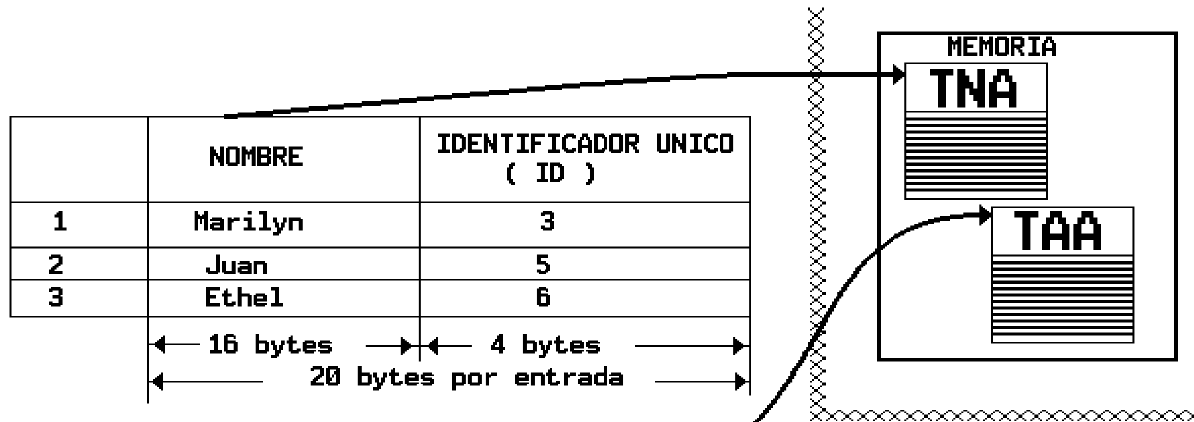
15.2.2 - **SISTEMA DE ARCHIVO SIMBOLICO**

El primer módulo que es llamado cuando hacemos un pedido al Sistema de Archivos es el módulo denominado Sistema de Archivo Simbólico (SAS).

Una típica llamada sería por ejemplo:

CALL SAS(READ,"JUAN",4,1200)

Donde estamos pidiendo que se lea el registro lógico número 4 del archivo "JUAN", para colocar su



	NOMBRE	IDENTIFICADOR UNICO (ID)
1	Marilyn	3
2	Juan	5
3	Ethel	6

a) Directorio de archivo simbólico

ID	LONGITUD DE REGISTRO LOGICO	CANTIDAD DE REGISTROS LOGICOS	DIRECCION AL PRIMER BLOQUE FISICO	DIRECTORIO Y CONTROL DE ACCESO
1	---	---	0	Dir. Básico
2	20	3	1	Dir. Simbólico
3	80	10	2	Todos Leen
4	1000	3	3	Libre
5	500	7	6	Todos Leen
6	100	30	12	MARTA Lee JUAN Lee/Graba
7	1000	2	10	Libre
8	1000	1	15	Libre

b) Directorio de archivo básico

Fig. 15.2. - Directorios Simbólico y Básico.

contenido en la dirección 1200 de memoria principal.

El Sistema de Archivo Simbólico usa el nombre del archivo para localizar la única entrada que posee el archivo en el Directorio de Archivos.

Ahora bien, si supusiéramos que existe una correspondencia biunívoca entre el nombre que puede tener un archivo y el archivo en cuestión, nos encontraríamos que el sistema adolece de una gran falla. Pues no podríamos referenciar a un archivo por diferentes nombres y distintos archivos no podrían tener el mismo nombre.

Para implementar tal facilidad dividimos el Directorio de Archivos en dos partes. Un Directorio de Archivo Simbólico (DAS) y un Directorio de Archivo Básico (DAB) como se muestra en la figura 15.2.

El Sistema de Archivo Simbólico debe buscar en el Directorio de Archivo Simbólico la entrada perteneciente al archivo requerido y de esta forma encontrar su único identificador (ID) dentro del sistema, para pasárselo al módulo denominado Sistema de Archivo Básico (SAB).

En nuestro ejemplo CALL SAS(READ,"JUAN",4,1200) devolvería 5 donde 5 es el ID de del archivo "JUAN".

Normalmente como el DAS es mantenido en el periférico de almacenamiento, lo que se hace es copiar en memoria las entradas del DAS que corresponden a archivos en uso (llamados archivos abiertos o activos), de tal forma que estas entradas son usadas para evitar E/S sobre la misma zona en el periférico. Esta tabla que se mantiene en memoria principal con las entradas del DAS correspondientes a archivos abiertos recibe el nombre de Tabla de Nombres Activos (TNA).

15.2.3 - **SISTEMA DE ARCHIVO BASICO**

El segundo módulo que es llamado en la secuencia se denomina Sistema de Archivo Básico (SAB). La llamada sería de esta forma:

CALL SAB(READ,5,4,1200)

Donde todos los parámetros son iguales a la llamada del módulo SAS, a excepción del segundo parámetro que constituye el identificador que le pasó el SAS.

El SAB se vale del identificador del archivo (ID) para localizar la entrada correspondiente a este en el Directorio de Archivo Básico.

Dicha entrada es mantenida en memoria principal con el objeto de ahorrar posteriores E/S buscando la misma entrada. La tabla que alberga todas las entradas del Directorio de Archivo Básico de los archivos abiertos recibe el nombre de Tabla de Archivos Activos (TAA). Esta tabla se genera y mantiene en memoria principal a diferencia del directorio básico (DAB) y del directorio simbólico (DAS). Su entrada consta de la información que puede visualizarse en la Fig. 15.3.

Identificación	Long Reg. lógico	Long. Reg. físico	Formato	Organización

Permisos	Concurrencia	Lista de Procesos que lo están usando

Fig. 15.3. - Tabla de Archivos Activos (TAA).

Para la próxima etapa - Verificación de Control de Acceso (VCA)- utilizaremos en vez del ID del archivo, su entrada correspondiente en la TAA, la cual contiene la información del archivo ID 5.

La invocación al módulo de Verificación de Control de Acceso es de la siguiente forma:

CALL VCA(READ,5,4,1200)

Donde 5 es la entrada 5 de la TAA que corresponde al archivo "JUAN" y los demás son exactamente los mismos parámetros de la llamada al módulo anterior.

En resumen podemos decir que el módulo SAB se encarga de:

1. El Directorio de Archivo Básico.
2. La Tabla de Archivos Activos.
3. La comunicación con el módulo VCA.

15.2.4. - **VERIFICACION DE CONTROL DE ACCESO**

Este módulo actúa como un punto de control entre el Sistema de Archivo Básico y el módulo de Sistema de Archivo Lógico, de tal forma que verifica si la función que se quiere realizar sobre el archivo en cuestión (READ, WRITE, etc.) está explicitada en la entrada correspondiente al archivo en la Tabla de Archivos Activos.

En caso que no sea permitido realizar dicha operación sobre el archivo estamos en presencia de una condición de error, por lo cual el pedido al Sistema de Archivos es abortado. Si efectivamente se puede realizar esa operación entonces el control pasa directamente al módulo de Sistema de Archivo Lógico.

15.2.5. - **SISTEMA DE ARCHIVO LOGICO**

Una llamada al módulo de Sistema de Archivo Lógico (SAL) posee la misma sintaxis que la llamada al módulo anterior. Luego, para el ejemplo que damos quedaría:

CALL SAL(READ,2,4,1200)

El Sistema de Archivo Lógico convierte el pedido de un registro lógico en el pedido de una secuencia de bytes lógicos, la cual se entrega al Sistema de Archivo Físico (SAF).

Esto es así, puesto que para el SAF un archivo no es más que una secuencia de bytes sin ningún tipo de formato.

Para nuestro ejemplo, vamos a suponer un formato de registro lógico de longitud fija, luego la conversión necesaria la podemos obtener de la información de la entrada en la TAA.

Dirección del byte lógico (dbl) = (número de registro - 1) * longitud del registro lógico = 3 * 500 = 1500

long. de la secuencia de bytes lógicos (lsb) = long. del registro lógico

Teniendo ya calculado el comienzo de la secuencia de bytes y su longitud, el SAL invoca al módulo de Sistema de Archivo Físico (SAF):

CALL SAF(READ,2,1500,500,1200)   1500 es dbl y 500 el lsb

Donde el tercer y cuarto par metro corresponden al dbl y el lsb respectivamente y el resto de parámetros son exactamente los mismos que los de la llamada el módulo anterior.

15.2.6. - **SISTEMA DE ARCHIVO FISICO**

El SAF tiene por función determinar en que bloque físico del dispositivo de almacenamiento se encuentra la secuencia de bytes lógicos pasados por el SAL, para lo cual se vale de la entrada en la TAA más el dbl y el lsb.

El bloque es entonces leído y colocado en un buffer preasignado en memoria principal, luego es extraído a partir de este buffer la secuencia de bytes pedidos y colocados en el área de buffer del usuario.

Para realizar estos cálculos el SAF debe saber las funciones de mapeo y longitud del bloque físico que utiliza cada periférico de almacenamiento. Si estas fueran las mismas para todo tipo de periférico podrían estar tranquilamente insertas en las mismas rutinas del SAF, pero tal cosa no sucede pues hay variaciones de un periférico a otro.

Esto se resuelve manteniendo tal información en el mismo volumen de almacenamiento, de tal forma que cuando se inicializa el sistema toda esta información pueda ser leída en una entrada de la Tabla de Archivos Activos. Si se hace un pedido de escritura y el bloque sobre el cual se quiere escribir no está asignado previamente entonces el Sistema de Archivo Físico invoca al Módulo de Estrategia de Asignación (MEA) para que este le proporcione un bloque libre en memoria secundaria sobre el cual pueda efectuar la escritura.

15.2.7. - **MODULO DE ESTRATEGIA DE ASIGNACION**

Este módulo se encarga de llevar un registro del espacio libre disponible en el almacenamiento secundario, tal información aparecerá reflejada en la TAA.

En la figura 15.B se muestra como se puede hacer uso del Directorio de Archivo Básico para llevar cuenta de esta información, es decir tratamos a los espacios libres en memoria secundaria como si fueran archivos. Obviamente existen otras técnicas más óptimas para tratar este problema.

15.2.8 - **MODULO DE ESTRATEGIA DE PERIFERICO**

Este módulo convierte el número de bloque físico en el formato adecuado para el periférico en cuestión (por ejemplo bloque 7 = pista 3 ,sector 23). Además de esto, inicializa los comandos adecuados de E/S para el tipo de periférico sobre el cual se va a realizar la operación.

Cabe destacar que todos los módulos anteriormente citados son totalmente independientes de cualquier tipo de periférico con excepción del último nombrado. De aquí en más, el control pasa a manos del Administrador de Entrada-Salida.

15.2.9. - **Resumen de los módulos**

A guisa de resumen reseñamos brevemente todos los módulos antes mencionados:

1) **Sistema de archivos simbólicos (SAS)**: transforma el nombre del archivo en el identificador único del Directorio de archivos. Utiliza la Tabla de nombres activos (TNA) y el directorio de archivos simbólico (DAS).

2) **Sistema de archivos básico (SAB)**: copia la entrada de la VTOC en memoria. Utiliza el directorio de archivos básicos (DAB) y la Tabla de archivos activos (TAA).
3) **Verificación de control de acceso (VCA)**: verifica los permisos de acceso al archivo.
4) **Sistema de archivo lógico (SAL)**: transforma el pedido lógico en un hilo de bytes lógicos.
5) **Sistema de archivo físico (SAF)**: calcula la dirección física.
6) **Módulo de estrategia de asignación (MEA)**: consigue espacio disponible en el periférico (casos de grabación).
7) **Módulo de estrategia de periférico**: transforma la dirección física según las características exactas del periférico requerido.

15.3. - ESTRUCTURAS DE DIRECTORIOS

Al hablar del Sistema de Archivos Básico hemos aclarado que el mismo opera sobre el Directorio de Archivos Básico (DAB). Usualmente este directorio se encuentra almacenado el algún periférico del sistema que posea velocidad alta.

Si pensamos un momento la cantidad de archivos que pueden llegar a existir en cualquier centro de cómputos (aún siendo pequeño) comprenderemos inmediatamente que contar con un único directorio básico en forma de una interminable tabla que deberá ser accedida toda vez que se requiera un archivo, resulta poco práctico.

En tal sentido es conveniente estructurar los directorios en forma jerárquica (similarmente a lo que se realiza con un archivo indexado con diferentes niveles de índices) para proveer una mejor y más óptima forma de acceso, un marco de trabajo ordenado y protección para todos los usuarios en su conjunto.

La cuestión de un marco de trabajo ordenado permite armar estructuras que por ejemplo pueden agrupar en un único directorio todos los archivos de un usuario específico, como así también todos los archivos y/o programas de una aplicación específica (sueldos, cuentas corrientes, etc.). La cuestión de cómo proteger dichas estructuras se verá más adelante (Ver punto 15.4).

15.3.1. - Directorios de un solo nivel

Una de las formas de estructurar un directorio puede verse en la Fig. 15.4. Nótese que en este caso la entrada en el directorio de archivos básico está apuntando a una estructura que mapea todos los bloques del archivo real en disco.

La cantidad de acceso necesarios para llegar a un archivo en una estructura de este tipo serán 2, uno para llegar al directorio básico y uno para llegar al archivo real. Si consideramos que para llegar a la tabla de mapas de bloques se requiere un nuevo acceso entonces la cantidad de acceso será del orden de 3.

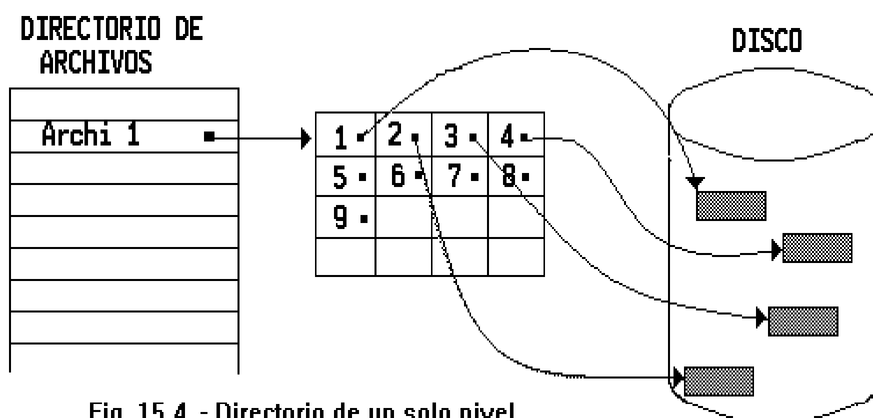


Fig. 15.4. - Directorio de un solo nivel.

La tabla de mapeo se utiliza para optimizar la utilización del espacio en el disco aunque cuenta con el inconveniente de que la información puede estar muy dispersa dentro del dispositivo degradando todo tipo de proceso que requiera leer una gran cantidad de registros de este archivo.

15.3.2. - Directorios de varios niveles

Una estructura más compleja de directorios puede visualizarse en la Fig. 15.5. En este caso la cantidad de accesos para llegar a un archivo dependerá de la profundidad de niveles del directorio.

En forma genérica podemos decir que para llegar a una entrada en el nivel n se requerirán no menos de n+1 accesos a disco, debido a que es necesario siempre recorrer los n niveles que apuntan a dicho archivo para finalmente acceder finalmente al archivo en sí.

Nótese en el ejemplo que para llegar al archivo ALFA se deberá ingresar al sistema por el directorio A, de allí pasar al segundo nivel en el directorio AA y de allí acceder al archivo; por lo tanto para proveer el camino completo para referenciarse al archivo un usuario debería especificar el nombre del archivo de la forma A/AA/ALFA pudiendo utilizarse como separadores la /, el ., o cualquier otro similar provisto por el lenguaje de control.

En las estructuras de este tipo debe tenerse presente que en las entradas de los directorios debe existir algún campo que permita diferenciar si la información contenida en esa entrada corresponde a un directorio o a un archivo (por ejemplo la entrada correspondiente al directorio ABA y la entrada correspondiente al archivo ALFA en el directorio AA segundo nivel).

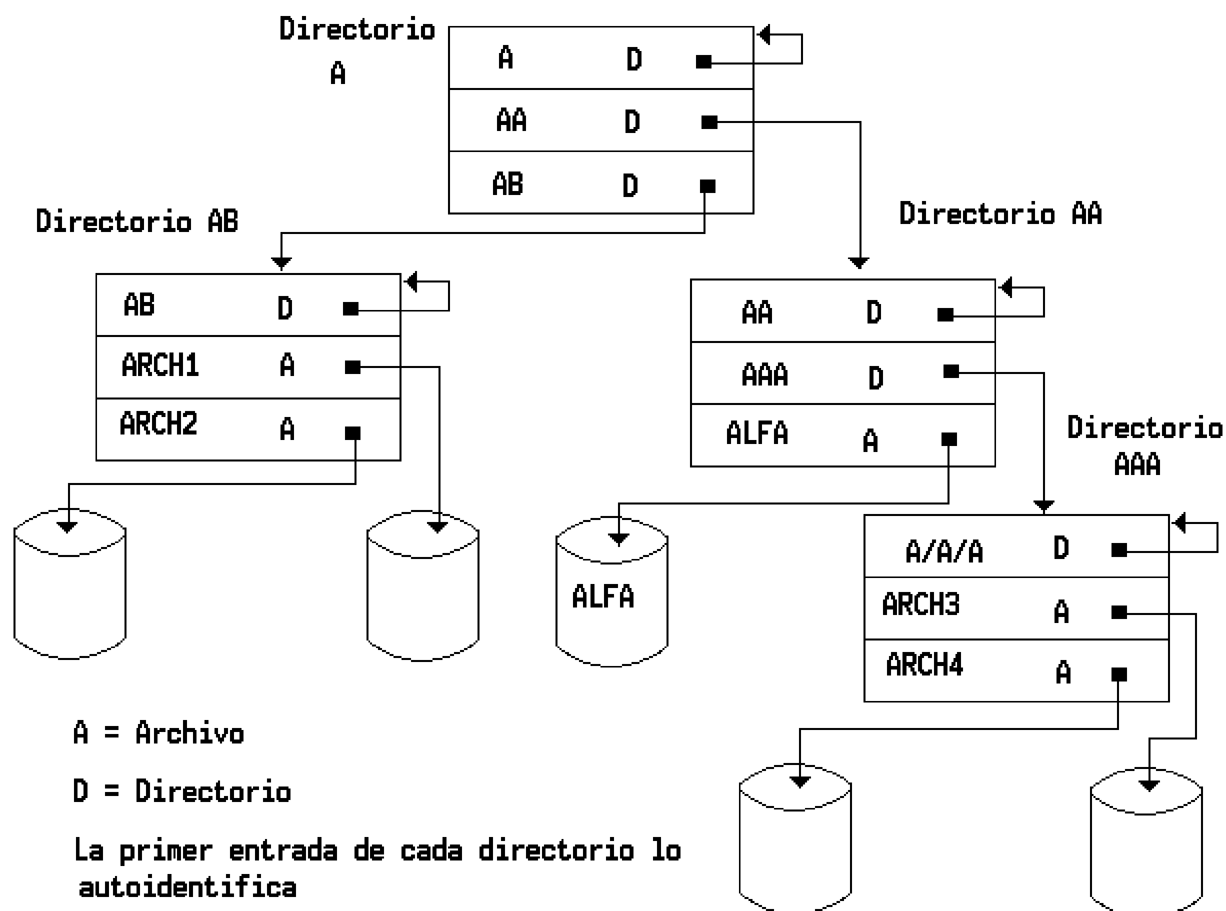


Fig. 15.5. - Directorios de varios niveles.

15.4. - **Estructuras de Control de Acceso**

El control de acceso a un archivo está dado por una serie de permisos que son controlados por el módulo de Verificación de Control de Acceso (VCA).
Tales permisos se almacenan conjuntamente con la información del directorio básico o alguno de sus niveles y pueden constituir asimismo archivos de permisos. Estos datos serán copiados por el Sistema de archivos Básicos en la TAA y luego consultados por el VCA para autorizar el acceso.
Una vez liberado el archivo y de haber ocurrido algún cambio en los permisos la información deberá ser devuelta a su lugar de almacenamiento en memoria secundaria.

15.4.1. **LISTA DE CONTROL DE ACCESO**

El concepto de la Lista de Control de Accesos (LCA) se basa en que para cada archivo en el sistema se asocia una lista de permisos (que puede ser otro archivo) en donde se especifica para cada usuario que tipo de acciones puede realizar sobre ese archivo.
En el ejemplo de la figura 15.6 el usuario JOSE puede Leer sobre el archivo ALFA y el usuario MARIA puede Leer/Grabar sobre el archivo ALFA.
Suele existir también el permiso de Owner que indica el nombre del usuario que originalmente creó ese archivo y que puede realizar todo tipo de acciones sobre él. En el ejemplo no se ha especificado este permiso debido a que para llegar al archivo ALFA se debe ingresar obligatoriamente por el directorio del usuario PEPE asumiéndose entonces que dicho usuario es el Owner.
Pueden existir también permisos de Ejecución cuando se trata de archivos que corresponden a códigos objeto de programas. Este permiso es muy útil ya que solamente permite el acceso a tal archivo si la acción de-

seada es Ejecutar y en cambio lo niega si la acción deseada es de Lectura (previene copias ilegales). La acción se puede diferenciar en virtud de la naturaleza del proceso que la invoca (un COPY es diferente de un EXEC).

Nótese que por la estructura que poseen el directorio de ejemplo es requisito indispensable que para que un usuario llegue a un archivo que no le es propio conozca el nombre del usuario que es su propietario a fin de

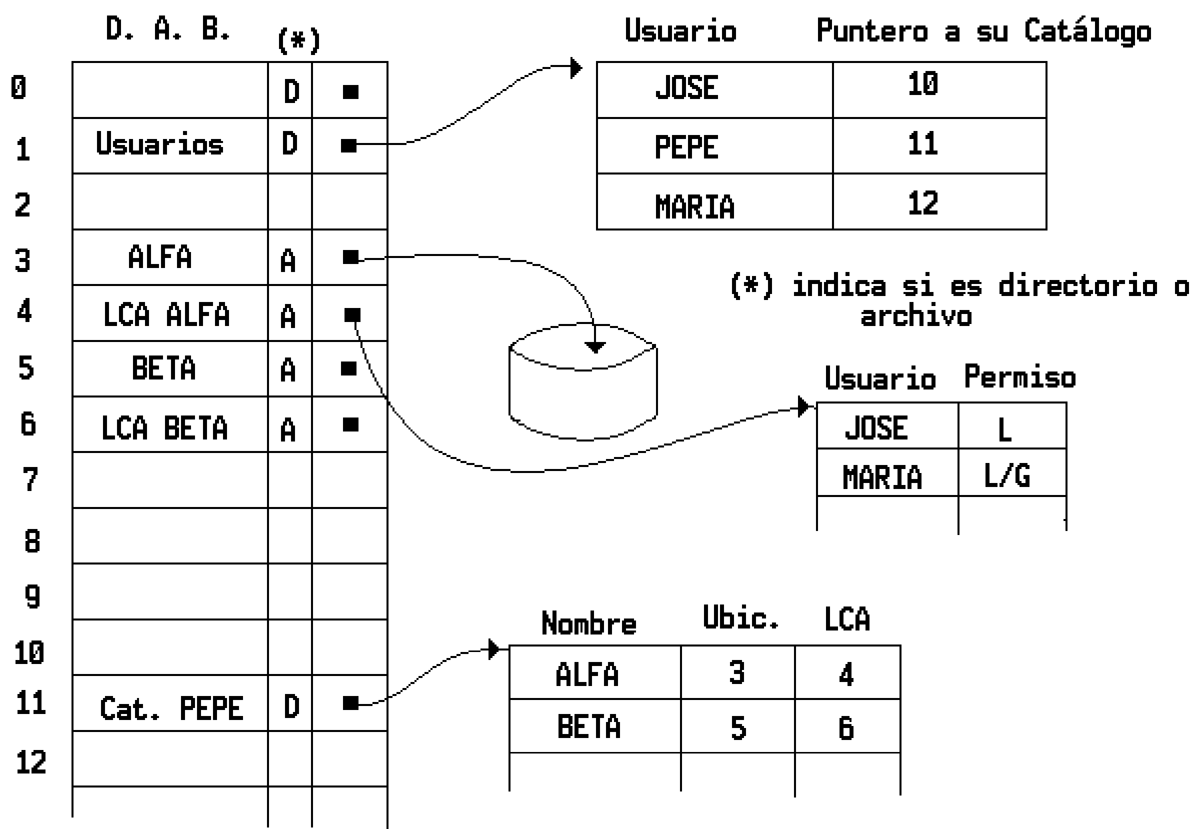


Fig. 15.6. - Sistema de Lista de Control de Acceso (LCA).

proveer la información de la vía necesaria para ubicar ese archivo (path). En tal estructura un acceso al directorio básico con el nombre ALFA no sería útil debido a que el módulo VCA exige la verificación de la LCA del archivo y tal información solamente existe a nivel del directorio del usuario PEPE (obvio, el acceso por defecto es Ninguno).

La forma más sencilla de proteger el directorio básico es aprovechar la entrada cero, que existe siempre en todo tipo de directorio y que sirve para autoidentificarse, para agregar allí la información de donde se encuentra su LCA.

En LCA la capacidad de acceso a un archivo pertenece al mismo archivo.

15.4.2. - **LISTA DE CONTROL DE USUARIO**

A diferencia de la LCA la Lista de Control de Usuario (LCU) se basa en el concepto de que para cada usuario existente en el sistema hay una lista de los archivos a los cuales puede acceder y que acciones puede realizar sobre ellos.

En la Fig. 15.7 podemos ver como con prácticamente la misma estructura de directorios anterior construimos una LCU.

Para cada usuario tenemos una entrada que corresponde al directorio de los archivos que le son propios y un puntero a una lista de archivos que indica la ubicación del archivo en el DAB, su nombre y el permiso de acceso.

Aquí puede verse inmediatamente como un mismo archivo puede ser visto con nombres diferentes por varios usuarios (JOSE/GAMMA y PEPE/ALFA son el mismo archivo).

En LCU la capacidad de acceso a un archivo pertenece al usuario que desea accederlo.

15.4.3. - **Comparación entre LCA y LCU**

Es siempre importante antes de comparar cualquier estructura LCA con la LCU tener en cuenta cómo es la estructura de directorios para acceder a los archivos ya que dependiendo de su complejidad y flexibilidad (pueden existir encadenamientos entre las entradas del DAB, encadenamientos entre las entradas de los archivos de los catálogos de usuarios, etc.) es que pueden sopesarse correctamente sus ventajas y/o desventajas.

Uno de los principales inconvenientes se presenta al querer deletear (borrar, eliminar) un archivo.

En la LCA la baja de un archivo no presenta mayores inconvenientes ya que se elimina la entrada del archivo en el catálogo del usuario Owner y luego se liberan las entradas correspondientes al archivo y a su LCA en el DAB.

En cambio en LCU presenta un inconveniente, ya que no se sabe a priori qué otros usuarios apuntan a ese archivo. Una solución es al eliminar el archivo recorrer **todas** las LCU de los otros usuarios rastreando aquellas cuyo puntero al básico coincida con el archivo eliminado para darlas de baja. Esta tarea se puede ver facilitada si se encadenan las entradas para un mismo archivo que correspondan a diferentes usuarios, pero por otra

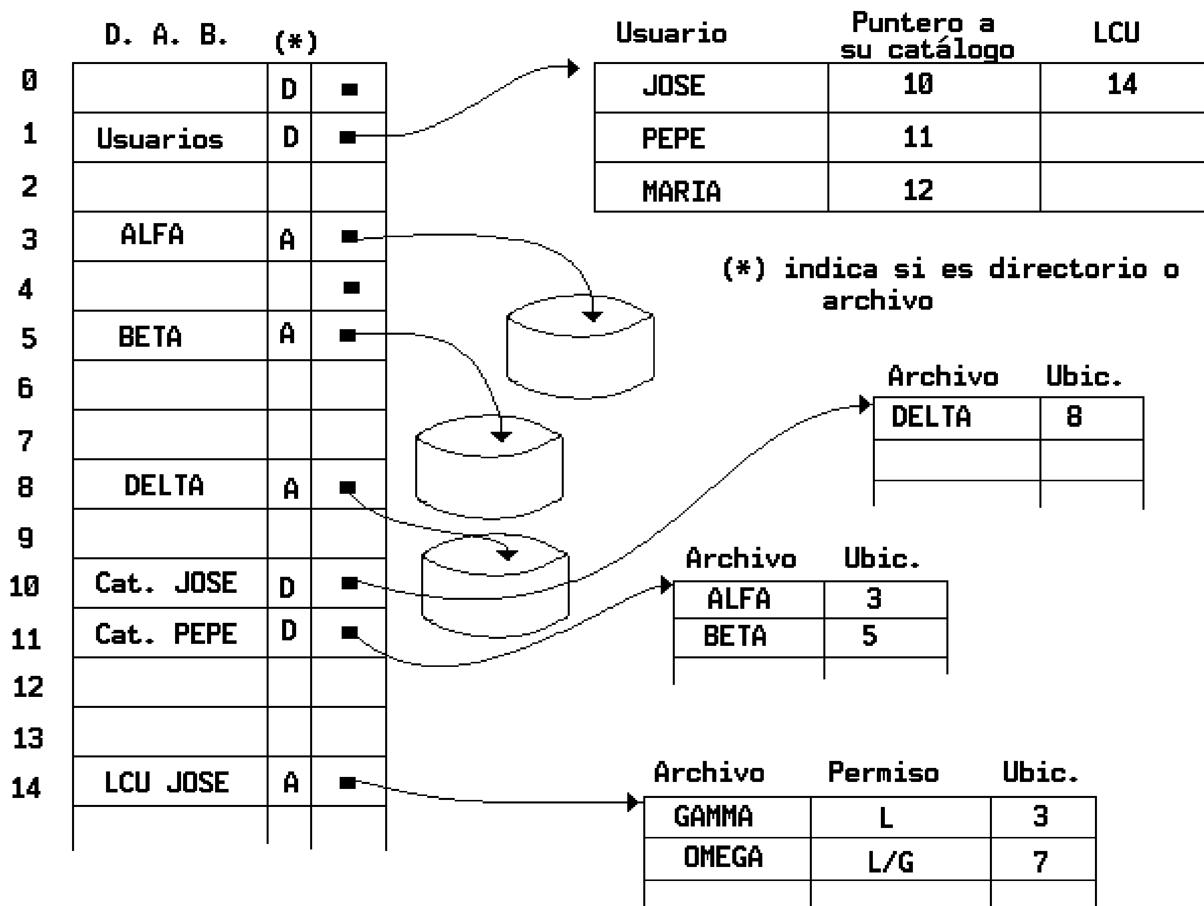


Fig. 15.7. - Sistema de Lista de Control de Usuario (LCU).

parte complica el mecanismo de mantenimiento y actualización de archivos.

La LCU por su naturaleza permite fácilmente la existencia de diferentes nombres para un mismo archivo lo que beneficia al sistema desde un punto de vista de seguridad y confidencialidad, hemos visto que esto no es tan sencillo en el sistema LCA.

No es fácil implementar un permiso genérico (del tipo Todos Leen) en una LCU ya que debería colocarse en la LCU de cada usuario la referencia al archivo deseado.

15.5. - <u>OPERACION DE OTRAS INSTRUCCIONES DE E/S.</u>

Hemos visto en detalle en párrafos anteriores las diferentes acciones que desencadena una instrucción de lectura (READ) en el sistema de administración de la información. A continuación detallaremos las acciones que se suceden con otras instrucciones típicas de E/S.

15.5.1. - <u>Instrucción OPEN</u>

Para que un archivo pueda abrirse, éste deberá ser asignado previamente. La asignación puede hacerse dentro del programa con la ejecución de la instrucción de apertura del archivo o separadamente (a nivel etapa).
La asignación propiamente dicha realiza los siguientes pasos :

- generar la entrada para el dispositivo apuntada desde el BCP.
- genera la entrada para el archivo apuntada también desde el BCP y asocia el nombre interno con el nombre externo.
- genera una entrada en la TNA.

En aquellos casos en que el lenguaje no provee una instrucción específica de apertura generalmente la primer instrucción de lectura o grabación del archivo es la que desencadena las acciones de la instrucción OPEN.

15.5.1.1. - **Apertura de archivo con asignación a nivel etapa (OPEN sin ASSIGN)**

Como ya se realizó la asignación, ya existe una entrada en la Tabla de Nombres Activos (TNA), con esta información se procede entonces a buscar el archivo en la Tabla de Archivos Activos (TAA). Si se encuentra, implica que el archivo estaba siendo usado por otro proceso y se controlan los permisos y concurrencia según los campos respectivos en la TAA, y luego se asocia el dispositivo y el archivo al proceso.

Si no se encuentra el archivo en la TAA se deberá generar una entrada en la misma accediendo al Directorio de Archivos Básicos (DAB) en el disco y copiando la entrada correspondiente al archivo en memoria (TAA). Para la búsqueda del DAB es posible que haya que recorrer los volúmenes. Se controlan los permisos en el disco (que también se copian en la TAA) y finalmente se asocia el dispositivo y el archivo al proceso.

Es importante tener en cuenta que si el archivo es de escritura (OUTPUT) y se lo utiliza por primera vez se debe solicitar la asignación del espacio libre al Módulo de Estrategia de Asignación.

15.5.1.2. - **Apertura de archivo con asignación de dispositivos**

Se genera una entrada para el dispositivo y otra entrada para el archivo en el BCP. Como el OPEN debe realizar la asignación, asocia el nombre interno con el nombre externo generando una entrada para ese archivo en la TNA

A partir de este punto la apertura se realiza igual que en el apartado anterior.

15.5.2. - <u>Instrucción CLOSE</u>

Se busca la entrada del archivo en la Tabla de Nombres Activos (TNA). Con el identificador que se extrae de la TNA, se accede a la TAA. Por el Sistema de Archivos Físicos se puede saber si hay algún bloque que deba ser grabado o no (solamente se chequea esto último si el archivo fue utilizado de output o de input-output). Si éste no fue grabado, se debe realizar una E/S física (en este caso un WRITE), se llama al Módulo de Estrategia de Asignación (MEA), luego al Módulo de Estrategia de Periférico y se lanza la E/S física.

Si el archivo se está usando concurrentemente con otros procesos se elimina el proceso de la lista de archivos asociados al proceso en la TAA y se decrementa en uno el campo concurrencia de la misma tabla.

Si el archivo no se usa concurrentemente, es decir el campo concurrencia es igual a 1, se verifica si su entrada en la TAA fue modificada (casos de output, input-output o información de fechas), por lo tanto se debe actualizar esta información en el disco. Con SAB se graba esta información de la TAA al Directorio de Archivos Básicos en el Disco y por último se borra la entrada del archivo de la lista de procesos que lo están usando y se borra la entrada del archivo en la TAA (concurrencia = 0).

15.5.3. - <u>Instrucción DELETE</u>

Para borrar un archivo se debe controlar si el usuario que desea borrar el archivo tiene autorización para hacerlo (VCA - Puede ser el dueño o tener permiso de borrado). Si no está autorizado, se produce un error y no se puede realizar la acción.

Caso contrario, si se trata de un esquema de protección de tipo LCU, se recorren la listas de los usuarios buscando aquellas que apuntan a la entrada del archivo en el Básico y se las elimina. Luego se elimina físicamente el archivo, devolviendo el espacio ocupado por el mismo como espacio disponible. La eliminación física del archivo implica la baja de la entrada del directorio básico que apunta a la dirección física real del archivo en cuestión.

El recorrido según las listas es el siguiente :

LCU

- Por cada usuario del sistema :
    - i - Ingresar en la LCU de los archivos que no le son propios.
    - ii - Eliminar el archivo de la lista
- Para el usuario dueño del archivo :
    - i - Elimina el archivo de la lista de archivos propios.

LCA

- i - Ingresar a la LCA del archivo y la elimina.
- ii - Devuelve también la entrada de la LCA como espacio libre.

Es necesario aclarar que ciertos lenguajes proveen la instrucción DELETE como una de las disponibles. En cambio en otros sistemas la acción de borrar un archivo se especifica por medio de una tarjeta de control. De todas maneras en la mayoría de los sistemas operativos de hoy en día se verifica la concurrencia de la acción de borrado respecto de la utilización del archivo por algún usuario en el momento en que se desea borrar el mismo a efectos de prevenir el fin anormal de una tarea.

15.6. - **ALGUNAS CONSIDERACIONES MAS: Sistemas de Archivo Lógico o Método de Acceso (Dirección lógica).**

El "cálculo de la dirección lógica" depende directamente de la estructura de los registros (formatos), de la organización del archivo y de la forma en que se quiere acceder a la información.

Las formas de llegar a la información, dependiendo de sus estructuras, organización y acceso es lo que conforman los **Métodos de Acceso**.

Luego, los Métodos de Acceso serán distintos (operarán de distinta manera) dependiendo de:
- la forma en que se quiere acceder a la información,
- del formato de los registros y
- la organización del archivo.

Veamos algunos ejemplos.

15.6.1. - **Archivo Secuencial, Formato Fijo.**

En este tipo de archivo los registros tienen igual longitud, como podemos ver en el siguiente grafo de la Fig. 15.8.

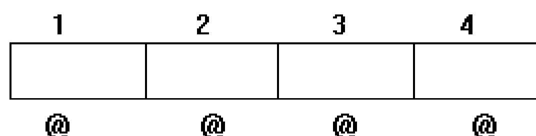


Fig. 15.8.

15.6.1.1. - **Acceso Secuencial**.

En la TAA se mantiene un NBI (Número de Byte Inicial) (dbli) y un NBF (Número de Byte Final) (dblf)
NBI = inicialmente está en cero
NBF = número siguiente al del último registro grabado.
Luego, en este caso el (SAL) Método de Acceso actúa de la siguiente forma:
En lectura, luego de procesar cada registro se cambia
NBI = NBI + LR (siendo LR la longitud del registro)
En escritura, si está permitido escribir luego del último registro (append), se cambia
NBF = NBF + LR

Nótese que el NBI debe estar en la TAA y si el archivo es compartido debe existir además un NBI para cada proceso que lo comparte.

El NBF es un dato que está incluido en la Tabla de Descripción del Archivo, y debe incluirse en la TAA durante la operación de OPEN. Obviamente habrá uno solo.

En el momento de la creación de un archivo NBI y NBF mantienen el mismo valor.

15.6.1.2. - **Acceso Directo**

Aquí se aplica el ejemplo ya visto, donde dado un NR (número de registro) se aplica:
NBI = (NR - 1) * LR

15.6.2. - **Archivo Secuencial, Formato Variable.**

En este tipo de archivos la longitud de cada registro es variable, de la siguiente forma:

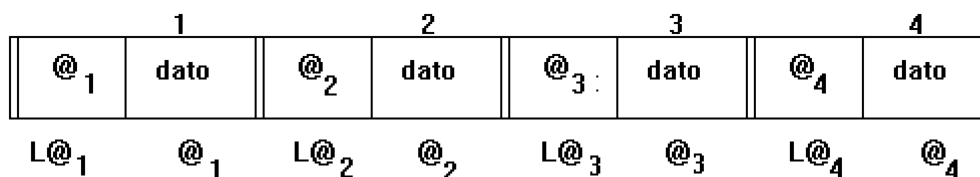


Fig. 15.9.

Donde
$L@_i$ = contiene la longitud del registro (sus longitudes son iguales)
$@@_i$ = contiene los datos en sí (sus longitudes son distintas)

15.6.2.1. - **Acceso Secuencial.**

Aquí valen las mismas consideraciones realizadas antes para NBI y NBF.
NBI = inicialmente en cero
luego el próximo registro se encontrará en :
NBI = NBI + $L@_n$ + $@_n$
donde

$L@_n$ = longitud del campo longitud del registro n
$@@_n$ = longitud del dato

Hay que tener en cuenta que es necesario mantener en memoria un buffer de longitud suficiente para albergar el registro más largo posible de este archivo, dato que está en la descripción del mismo archivo.

15.6.2.2. - **Acceso Directo.**

Si se quiere acceder al Registro Número 3, la forma de obtener su dirección es únicamente posible por medio de :

$NBI = L@_1 + @_1 + L@_2 + @_2$

Obviamente acceder a un registro de esta manera es muy ineficiente.

Algunas maneras de mejorar esta situación serían :

1)- Mantener el valor del último NBI de tal manera que si el próximo registro que se busca es superior al anterior buscado, se comienza desde ese.

2)- Mantener el valor de todos los NBI leídos para usos futuros (Note/Point) (con la esperanza de reusarlos)

3)- Mantener en memoria una tabla para todos los NBI (tiene la desventaja de una gran ocupación de espacio en memoria)

Para otras organizaciones será necesario que el Método de Acceso consulte las estructuras propias de esa organización.

Por ejemplo, en un Secuencial Indexado, dada su clave se deberá encontrar en los índices el valor del NBI; en un Random con clave, de deberá calcular a través de la clave y una función (si es provista) el valor del NBI.

A medida que las estructuras son más complejas y se brindan mayores facilidades de búsqueda (a través de relaciones, etc.) dentro del mismo archivo, e inclusive entre distintos archivos (referencias cruzadas, etc.) se encuentra que los Métodos de Acceso se transforman en Sistemas de Gestión de Bases de Datos.

15.6.2.3. - **Sistemas de Archivos Físico (SAF).**

El cálculo de la dirección física se realiza de la siguiente manera:

Número de Bloque Relativo (NBR) = [DBL / Longitud Bloque]   y

DF = NBR + Dir. Archivo (o dirección del 1er Bloque)

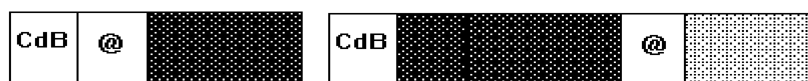
Además se ubica la posición de la información dentro del Bloque (byte dentro del bloque) como :

Posición Relativa Registro (PRR) = Resto [DBL / Longitud Bloque]

Si DF es una dirección cuyo contenido ya se encuentra dentro del buffer en memoria esto significa que **no** es necesario realizar una operación de E/S física, sólo es necesario trasladar la información al área del proceso (o pasar la dirección de la información), en caso contrario se sigue adelante con el resto de las funciones de la Administración de Archivos y las operaciones de E/S físicas.

Nótese aquí que si se estuviera trabajando con un Sistema de Administración de Memoria Paginada por Demanda, y el buffer no se encontrase en memoria se produciría una Interrupción por Falta de Página.

Pueden existir situaciones especiales como el caso de un registro lógico que supere el tamaño de los registros físicos, como por ejemplo :



CdB: Cabecera de bloque (usualmente 4 bytes de longitud, indica la longitud del bloque)

Fig. 15.10.

Aquí habría que determinar cuantos bloques es necesario traer a memoria, que se calcula como:

Cantidad de bloques = [Long. Lógica / Long. Bloque] + 1

si PRR = 0, o sea el registro comienza al comienzo del bloque y

Cantidad de bloques = [Long. Lógica / Long. Bloque] + 2

si PRR ¹ 0, o sea que el registro comienza en el interior de un bloque.

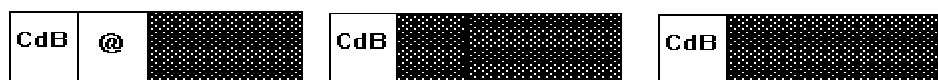


Fig. 15.11.

Una vez que la información está en el buffer se trasladan @ bytes al área del usuario.

15.7. – **FILE SYSTEMS DE EJEMPLO**

15.7.1. - **File System en UNIX**

Se utiliza como base el UNIX 4.3BSD (Berkeley Software Distribution) en máquinas VAX (1987).

15.7.1.1. - **Manejo de Archivos**

Para algunos sistemas operativos los archivos son conocidos como estructuras, en UNIX sin embargo un archivo es una secuencia de bytes, el kernel no conoce estructuras.
Los archivos se organizan en estructuras llamadas directorios. Los directorios son archivos que tienen información para encontrar a otros archivos.
Un path name a un archivo es un string que indica el camino a través de las estructuras de directorios hacia un archivo. Se lo denota separando cada nivel con una “/”.

Pathname absolutos : empiezan en la raíz
Pathname relativos : empiezan en el directorio actual

Un archivo puede tener distintos nombres. Estos se conocen como links.
Hay dos tipos de links, los soft links tienen el path absoluto al archivo y pueden apuntar a directorios y a archivos en otros file systems.
En cambio los hard links no pueden apuntar a directorios ni a files en otros file systems.

“.” es un hard link al directorio actual
“..” es un hard link al directorio padre del directorio actual

Los dispositivos de hardware tienen nombres en el file system.
Estos archivos especiales de dispositivo o archivos especiales son conocidos por el kernel como interfases de dispositivos y son accedidos por el usuario con las mismas llamadas al sistema que para acceder a otros archivos.

<table>
<tr><td rowspan="9">/ Raíz</td><td>vmunix</td><td colspan="2">Imagen binaria de booteo del UNIX</td></tr>
<tr><td>dev</td><td colspan="2">Archivos especiales de dispositivos. Por ej. /dev/console, /dev/lp0, /dev/mt0</td></tr>
<tr><td>bin</td><td colspan="2">Archivos esenciales del UNIX</td></tr>
<tr><td>Lib</td><td colspan="2">Archivos de bibliotecas de C, Pascal, subrutinas FORTRAN</td></tr>
<tr><td rowspan="2">Usr</td><td>Bin</td><td>Programas de aplicación del sistema (editores de texto)</td></tr>
<tr><td>Local</td><td>Bin: programas del sistema escritos localmente</td></tr>
<tr><td>user</td><td colspan="2">Para usuarios</td></tr>
<tr><td>etc</td><td colspan="2">Archivos administrativos, por ej. Password</td></tr>
<tr><td>tmp</td><td colspan="2">Temporarios</td></tr>
<tr><td colspan="4">EJEMPLO DE CONTENIDOS DE LOS DIRECTORIOS EN UNIX 4.3 BSD</td></tr>
</table>

Las llamadas básicas del sistema para manipular archivos son :

- *Creat* : dado un path crea un archivo vacío o trunca uno ya existente.
- *Open* : abre un archivo. Dado un path y un modo (read, write o r/w) devuelve un entero pequeño que se denomina file descriptor. Este descriptor se pasa a la llamada read o write junto con la dirección del buffer y la cantidad de bytes a transferir.
- *Close* : se cierra un archivo cuando se pasa a esta llamada el file descriptor de ese archivo.
- *Trunc* : reduce la longitud de un archivo a cero
- *Lseek* : permite resetear explícitamente la posición dentro del archivo
- *Dup* o *dup2* : crea copias del file descriptor
- *Fcntl* : hace lo mismo que dup pero además puede ver o setear algunos parámetros de un archivo abierto, por ejemplo puede hacer que cada write al archivo realice un append.
- *Ioctl* : se usa para manejar parámetros de los dispositivos.
- *Stat* : devuelve información sobre el archivo por ejemplo tamaño, dueño, permisos, etc.
- *Rename* : cambia el nombre del archivo
- *Chmod* : cambia los permisos del archivo
- *Chown* : cambia el owner del archivo
- *Link* : crea un hard link para un archivo existente con un nuevo nombre
- *Unlink* : remueve un link. Si es el último el archivo se borra
- *Symlink* : crea un link simbólico
- *mkdir* : crea un directorio

- rmdir : borra un directorio
- *cd* : cambia de directorio

Un file descriptor es un índice a una tabla de archivos abiertos por este proceso. Van de 0 a 6 o 7 dependiendo de la cantidad de archivos abiertos que tenga el proceso.

Cada read o write actualiza el desplazamiento dentro del archivo que se encuentra asociado con la entrada en la tabla de archivos y se usa para determinar la posición en el archivo para el próximo read o write.

Las instrucciones para operar sobre directorios tienen que ser distintas de las anteriormente mencionadas, ya que la estructura interna de un directorio debe ser preservada.

Estas son : opendir, readdir, closedir, etc.

15.7.1.2. - **FILE SYSTEM**

UNIX soporta principalmente dos tipos de objetos: Archivos y Directorios. Los directorios son archivos con un formato especial.

15.7.1.2.1. - **Bloques y Fragmentos**

La mayoría del file system se mantiene en bloques de datos. Un tamaño de bloque mayor a 512 bytes es bueno por cuestiones de velocidad pero como en UNIX usualmente existen archivos de tamaño pequeño los bloques muy grandes causarían demasiada fragmentación interna.

Una solución consiste en utilizar 2 tamaños de bloques: Todos los bloques de un archivo con de un tamaño suficientemente grande, por ejemplo 8K, y el último bloque es un múltiplo de un pequeño fragmento, por ejemplo 1024 bytes.

Ejemplo: Un archivo de 18.000 bytes tendría 2 bloques de 8K y un fragmento de 2K.

Los tamaños de los bloques y fragmentos se especifican en la creación del file system.
Detalles de la implementación fuerzan a una relación bloque:fragmento de 8:1 y bloques con un mínimo de 4K, por ejemplo relaciones del estilo 4096:512 o 8192:1024 son válidas.

A medida que se generan un fragmento se copia en disco y si se genera otro fragmento entonces se copia y se junta el primer fragmento con el segundo, por lo tanto pueden llegarse a realizar 7 copias antes de obtener un bloque. Para evitar esto se proveen mecanismos para que los programas descubran el tamaño del bloque y evitar la recopia de fragmentos.

15.7.1.2. - **Inodos**

Un archivo se representa a través de un inodo (o i-nodo). Un inodo es un registro que almacena la mayoría de la información específica de un archivo en disco.

Inodo deriva de las palabras "nodo índice", antiguamente se escribía como i-nodo pero con el uso se reemplazó por inodo.

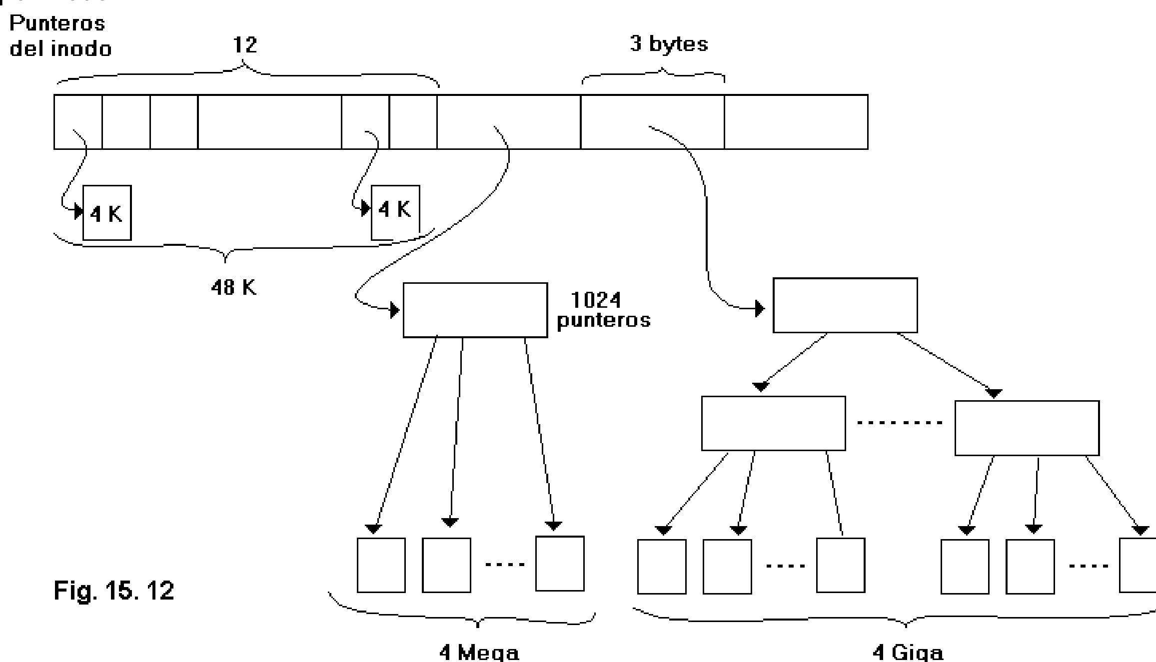


Fig. 15. 12

Un inodo contiene :
- identificadores del usuario y del grupo,

- hora de la última modificación y acceso al archivo
- un contador de hard links que apuntan al archivo
- tipo de archivo (plano, directorio, link simbólico, dispositivo de caracteres, disposit6ivoi de bloques o socket)

El inodo tiene 15 punteros a bloques de disco que contienen los datos. Los primeros 12 apuntan directamente a bloques. Si el archivo no es mayor a 12 bloques se puede acceder directamente por que el inodo se mantiene en memoria mientras el archivo está abierto.

Los otros 3 punteros usan direccionamiento indirecto. Los bloques apuntados así son de tamaño de bloque no de fragmento que se aplica solo a datos.

El primero de los 3 punteros es una indirección directa, el 2do es una doble indirección y el 3ero es una triple indirección aunque en general no llega a ser necesaria esta última ya que con bloques de 4K alcanza con doble indirección para un archivo de hasta 2^{32} bytes (4 Gigabytes).

En UNIX System V los bloques son de 1K y por lo tanto el máximo con doble indirección llega a 65 MB y con triple indirección llega a 16 GB.

El desplazamiento dentro de un archivo se maneja en memoria con una palabra de 32 bits. Los archivos, sin embargo, no pueden ser mayores a 2^{32} bytes.

15.7.1.3. - **DIRECTORIOS**

A este nivel no hay diferencia entre archivos planos y directorios. El contenido de los directorios se mantiene en bloques y se los representa con un inodo como a un archivo.

La única diferencia es el campo tipo en el inodo.

Los nombres de archivos son de longitud variables hasta 255 caracteres y por lo tanto las entradas en los directorios son de longitud variable. Cada entrada tiene primero su longitud, luego el nombre del archivo y el número de inodo.

Los primeros 2 nombres en todo directorio son ".”y “..” Las búsquedas en el directorio se hacen linealmente.

El usuario se refiere a un archivo a través de su path-name, sin embargo el file system usa el inodo y su definición del archivo, luego el kernel tiene que mapear el pathname que el usuario dio con el inodo. Para ello se usan los directorios.

Se parte desde el primer calificador del path (si es / es el raíz) y se obtiene su inodo. Se chequea el tipo de inodo y los permisos y se continúa con cada uno de los calificadores siguientes hasta obtener el inodo del archivo.

Los hard links se tratan como cualquier otra entrada del directorio.

Con los links simbólicos se comienza la búsqueda con el pathname que contenga el link y se previenen loops infinitos poniendo un límite de 8 links simbólicos.

Los archivos que son archivos de discos, por ejemplo los dispositivos, no tienen bloques de datos alocados en el disco. El kernel los reconoce por el tipo en el inodo e invoca a los drivers apropiados para manejar su E/S.

Una vez que se ubicó al inodo (por ejemplo a través de un open), se aloca una estructura de archivo que apunta al inodo. El descriptor de archivo que se entrega al usuario apunta a esta estructura de archivo.

15.7.1.4. - **Mapeando un descriptor de archivo con un inodo**

Las llamadas al sistema que referencian a archivos abiertos indican el archivo pasando un descriptor de archivo como argumento.

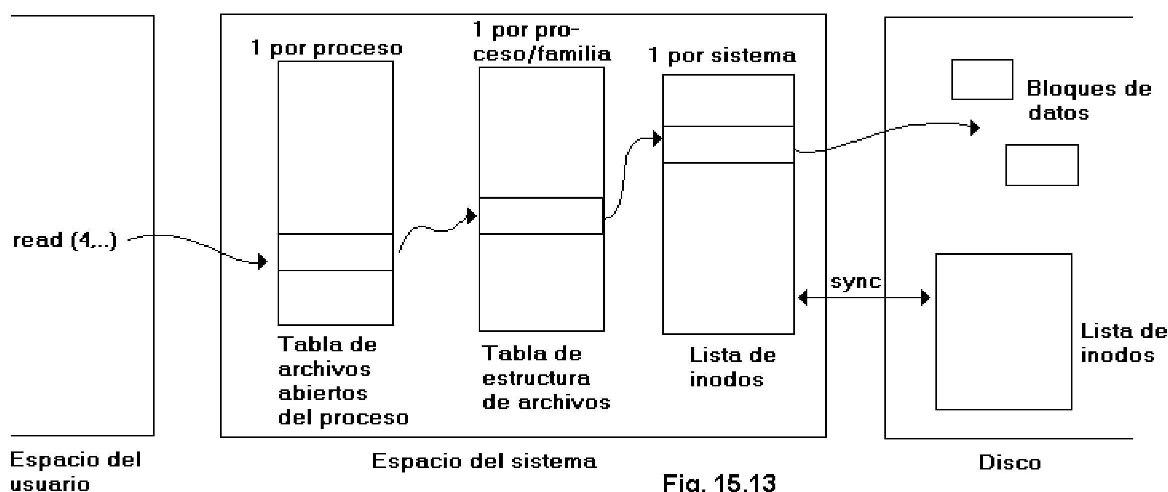


Fig. 15.13

El descriptor es usado por el kernel; para indexar una tabla de archivos abiertos por ese proceso. Cada entrada en la tabla tiene un puntero a una estructura de archivo. Esta estructura apunta al inodo.

Diferentes nombres de archivos pueden estar asociados con un mismo inodo pero un inodo que está activo está asociado exactamente con un solo archivo y cada archivo es controlado exactamente por un solo inodo.

El kernel mantiene un desplazamiento dentro del archivo que se actualiza con cada read o write.

Ya que más de un proceso puede estar usando concurrentemente un mismo archivo no es práctico llevar el offset en el inodo y por ende se lo mantiene en la estructura de archivo.

Las estructuras de archivos se heredan cuando un proceso padre crea un hijo (instrucción fork) y por lo tanto distintos procesos pueden tener el mismo offset dentro del archivo.

La estructura de inodos apuntada desde la estructura de archivos es una copia en memoria de la lista de inodos en el disco. Esta estructura se mantiene en una tabla de longitud fija.

La tabla en memoria tiene algunos campos extra como ser un contador de cuántas estructuras de archivos la apuntan y la estructura de archivos tiene un campo similar sobre cuántos descriptores de archivos la apuntan.

15.7.1.5. - **Estructura de Discos**

El usuario usualmente conoce solo un file system pero un file system lógico puede contener varios file systems físicos.

Usualmente se particionan los discos físicos en múltiples dispositivos lógicos, Cada dispositivo lógico define un file system.

Las ventajas de este enfoque son :

- distintos file systems pueden soportar diferentes usos
- es más confiable ya que una falla de software daña solo a ese file system
- cada file system puede tener diferentes tamaños de bloques/fragmentos
- evita que un solo usuario consuma el espacio total de los dispositivos físicos ya que está limitado al tamaño del file system

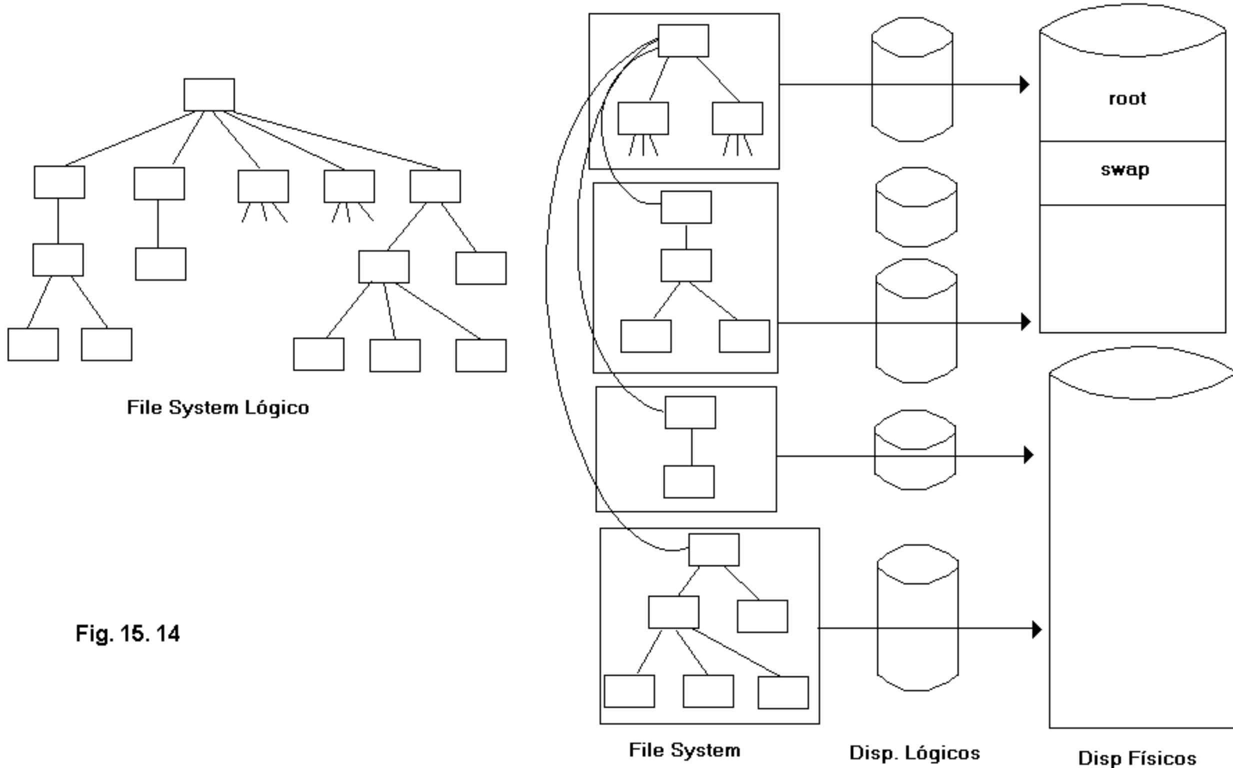


Fig. 15. 14

El file system raíz siempre está disponible y los otros pueden montarse, es decir se integran en la jerarquía del raíz.

Hay un bit en el inodo que indica si sobre él se ha montado un file system Una referencia a este archivo provoca que la tabla de montaje sea recorrida para ubicar el número de dispositivo montado.

El número de dispositivo se usa para encontrar el inodo del directorio raíz del file system montado y ese inodo es el que se utiliza.

De forma similar si un pathname es “..” y el directorio a buscar es el raíz de un file system montado, se recorre la tabla de montaje para encontrar el inodo sobre el cual se montó y se utiliza ese.

15.7.2. - **Sistema de Archivos de DOS**

15.7.2.1. - **Un poco de historia**

En 1977 Microsoft crea un sistema de archivos y lo llama FAT (File Allocation Table). Luego adaptaría la FAT para el sistema operativo DOS (Disk Operating System).

Originalmente el DOS fue pensado para ser usado desde un disco flexible, es por eso que la primer versión de FAT (FAT-12) puede almacenar como máximo 8Mb de información.

Este error de diseño es solo comparable con el error de diseño respecto al direccionamiento de la memoria de los procesadores de la línea 80x86 ("640k deben ser suficientes").

Viendo que 8Mb era poco Microsoft decidió dar un salto en tecnología y diseñar FAT-16, ahora la FAT podría almacenar los inalcanzables 32Mb!

En 1987 nuevamente los 32Mb ya no igualaban el tamaño de los discos y entonces se decidió atacar el problema de manera definitiva.

Viendo que 32Mb era poco se diseñó FAT-32, ahora la FAT podía almacenar 2Gb.

En 1995 ya existían algunos discos mayores a 2Gb.

Entonces era hora de aumentar nuevamente el tamaño de la FAT.

Viendo que 2Gb era nuevamente poco se diseñó VFAT (o FAT-32 extendido), ahora la FAT podía almacenar 2Tb.

Resumiendo:

Versión DOS	Nombre de FAT	Cantidad de Clusters	Cantidad Sectores por Cluster	Tamaño máximo de un archivo y de una partición
DOS 2.0	FAT-12	4Kb	4	8Mb
DOS 3.3	FAT-16	16Kb	4	32Mb
DOS 4	FAT-32	65Kb	64	2Gb
DOS 7(Win95)	VFAT (FAT-32 ext.)	4Gb	64	2Tb

En la actualidad el sistema de archivos de Windows 2000 soporta 64Tb, por ahora parece ser suficiente.

15.7.2.2. - **Introducción**

A continuación se explicara la estructura de las FAT-12 a FAT-32. Dado que cada nueva FAT es compatible con la anterior se tomara como base la FAT-12 agregando la información necesaria para extender los conceptos a FAT-16 o 32.

15.7.2.2.1. - **Algunas definiciones**

El ***sector*** es la unidad mínima de almacenamiento de los discos. Un sector tiene 512 bytes.

Un ***cluster*** es un conjunto contiguo de sectores que se agrupan para manipular mayor cantidad de información al mismo tiempo. Cada cluster tiene al menos 1 sector.

Cada archivo puede ocupar 1 o mas clusters, no necesariamente contiguos.

FAT es una sigla en ingles que significa "Tabla de ubicación de archivos".

15.7.2.2.2. - **La estructura de la FAT en DOS**

Si se tiene una FAT-16 se dispondrá de 65k clusters disponibles para almacenar 65k archivos o directorios en un principio.

Sin embargo el DOS utilizara algunos de ellos para almacenar información del sistema operativo. Ademas de estos clusters hay otros reservados para otros usos que no pueden ser usados por archivos del usuario.

Cluster Numero	Contenido
Cluster 0	Reservado para DOS
Cluster 1	Reservado para DOS
Cluster 2	2 (usado para almacenar un archivo chico)
Cluster 3	4 (usado para almacenar datos, extendido al cluster 4)
Cluster 4	5 (usado para almacenar datos, extendido al cluster 5)
Cluster 5	7 (usado para almacenar datos, extendido al cluster 7)
Cluster 6	0 (vacío, disponible para uso)
Cluster 7	FFFh (usado para almacenar datos, el ultimo "eslabón" de la cadena)
Cluster 8	0 (vacío, disponible para uso)
Cluster 65524	0 (vacío, disponible para uso)
Cluster 65525	0 (vacío, disponible para uso)
Cluster 65526	0 (vacío, disponible para uso)

15.7.2.3. - Capas lógicas en un disco FAT

Los sectores de un disco FAT están divididos en estos grupos, están listados en orden creciente al numero de sector.

1. Sectores reservados
2. FAT's
3. Directorio Raíz
4. Area de datos y Sectores ocultos

15.7.2.3.1. - **Sectores reservados**

En el sector lógico 0 se encuentra el bootsector [sector de arranque]
Contiene 512 bytes con la siguiente información.

Byte Nro.	Contenido	Descripción
0-2	Primera instrucción de la rutina de arranque	
3-10	Nombre OEM	Normalmente la versión de DOS, opcional.
11-12	Cantidad de bytes por sector	Múltiplos de 512
13	Cantidad de sectores por cluster	
14-15	Cantidad de sectores reservados	
16	Cantidad de copias de la FAT	Usualmente 2
17-18	Cantidad de entradas en el directorio raíz	
19-20	Cantidad total de sectores	
21	Descriptor del dispositivo	Disco 3½ / Disco 5¼ / Disco Rígido / Etc.
22-23	Cantidad de sectores en cada copia de la FAT	
24-25	Cantidad de sectores por pista	
26-27	Cantidad de lados	
28-29	Cantidad de sectores ocultos	
30-509	Rutina de arranque e información de la partición	
510	55(en hexadecimal)	
511	AA(en hexadecimal)	

15.7.2.3.2. - **FAT's**

En los sectores siguientes inmediatos se encuentran las copias de la FAT.
La Segunda copia de la FAT solo se utiliza para restaurar la 1ra en caso de necesidad.
Una FAT se almacena como una secuencia de registros de igual tamaño.
El tamaño del registro lo determina la versión de la FAT.
Para la FAT-12 el registro es de 12 bits. Para la FAT-16 en cambio es de 16 bits.

Cada registro codifica la siguiente información:

12-bit	16-bit	Significado
000	0000	Cluster disponible
001	0001	Entrada invalida
002-FEF	0002-FFEF	Cluster asignado, el contenido es el nro. del próximo cluster del mismo archivo en el mismo directorio
FF0-FF6	FFF0-FFF6	Reservado
FF7	FFF7	El cluster contiene sectores defectuosos
FF8-FFF	FFF8-FFFF	Cluster final de un archivo.

Dado que un archivo es una **lista encadenada de clusters**, el tamaño del archivo esta limitado por el tamaño de la partición FAT.

15.7.2.3.3. - **Directorio Raíz**

El directorio raíz contiene un registro de 32 bytes por cada archivo que esta en el.
Tiene una entrada mas para la etiqueta del volumen.

15.7.2.3.4. - **Area de datos y sectores ocultos**

La cantidad total de sectores codificadas en los bytes 19 y 20 del sector de arranque incluyen todas las áreas salvo la de sectores ocultos.

La cantidad total de sectores se calcula sumando a esa cantidad la cantidad de sectores ocultos que figuran en los bytes 28 y 29.

No se conoce un uso especifico para los sectores ocultos. Podrían usarse para "camuflar" otra partición o también para almacenar sectores con formatos especiales. Usualmente son pocos.

15.7.2.4. - **Estructura de directorios**

Cada entrada de directorio tiene la misma estructura:

Byte Nro.	**Contenido**
0-7	Nombre del archivo u 8 primeras letras del nombre del volumen
8-10	Extensión del archivo o 3 ultima letras del volumen
11	Byte de atributo*
12-21	No usado
22-23	Hora (bit 15-11 horas [0-23]; bit 10-5 minutos [0-59]; bit 4-0 "doble" segundos [0-29])
24-25	Fecha
26-27	Dirección del primer cluster
28-31	Cantidad de bytes del archivo. (Si es un subdirectorio o etiqueta de volumen vale 0)

*Significado del byte de atributo

Bit Nro.	Significado (bit = 1 = ON)	Bit No	Significado (bit = 1 = ON)
0	Solo lectura	4	La entrada representa un subdirectorio
1	Archivo oculto	5	El archivo fue modificado desde su ultimo backup
2	Archivo de sistema	6	No usado
3	La entrada representa una etiqueta de vol.	7	No usado

Si una entrada representa un directorio se setea en bit 4 y el resto no es usado.

Si una entrada representa una etiqueta de volumen se setea el bit 3 y el resto no es usado.

DOS no es case-sensitive, es decir no distingue las mayúsculas de las minúsculas para los nombres de archivo.

Campo fecha:

Bits 15-9: Años desde 1980 (0 a 127)
Bits 8-5: Mes (1 = Enero…12 = Diciembre)
Bits 4-0: Día (1 a 31)

Por lo tanto el "mundo DOS" empezó el 1 de enero de 1980 (00:00:00) y terminara el 31 de diciembre de 2107 (23:59:58).

El 1er carácter de una entrada de directorio tiene un significado especial para FAT:
Si es 00h significa que es el ultimo registro.
Si es 05h significa que la primer letra del nombre es el valor ASCII de E5h.
Si en cambio el valor es E5 significa que el directorio esta borrado.

Cuando un directorio es borrado su primer byte es reemplazado por el valor hexadecimal E5 para marcar que el directorio ha sido borrado.

El resto de la cadena no es modificada.

Esto es bueno por un lado porque:

- Permite la recuperación de archivos borrados.
- La operación de borrado es relativamente rápida.

Sin embargo presenta los siguientes problemas:

- El borrado genera fragmentación de los sectores.
- Es posible obtener nuevamente la información borrada lo que representa una falla de seguridad.
- Como lo único que se borra realmente del archivo es la 1er letra (esta fue reemplazada por E5) es necesario "conjeturar" este dato al momento de la recuperación del archivo.

15.7.3. - **HPFS**

15.7.3.1. - **Introducción**

Este sistema de archivos fue utilizado por primera vez en el sistema operativo OS/2 v1.2 para solucionar todos los problemas que la FAT no podía resolver. Las versiones 1.0 y 1.1 utilizaban FAT.

HPFS es una sigla en ingles (High Performance File System) que significa “Sistema de archivos de alta performance”.

15.7.3.2. - **Estructura de un volumen HPFS**

Un volumen HPFS utiliza un sector de 512 bytes y tiene un tamaño máximo de 2199GB.

Los discos removibles pueden usar HPFS, pero usualmente usan FAT pues:

- Es mas fácil transportar los archivos entre FAT, HPFS y otros sistemas mas fácilmente.
- Dado el tamaño de los discos removibles no se aprecian las mejoras de HPFS.

Notar que una de las estrategias de HPFS es mantener los datos tan contiguos como sean posibles, sin desperdiciar espacio para lograr eso.

15.7.3.2.1 - Estructuras fijas de un volumen HPFS:

Sector Nro.	Nombre	Descripción
0-15	BootBlock	32bit (Vol. Id) y programa de arranque.
16	SuperBlock	Punteros a: espacio libre, lista de sectores defect., directorio raíz y directorios. Fecha de ultima chequeo del volumen (CHKDSK/F)
17	SpareBlock	Punteros y Flags

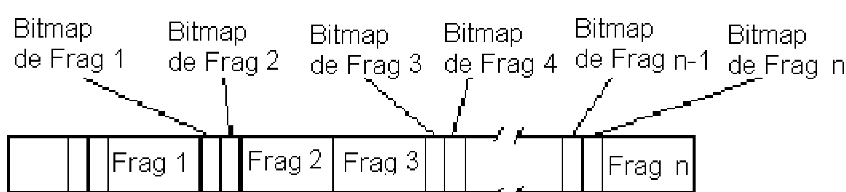
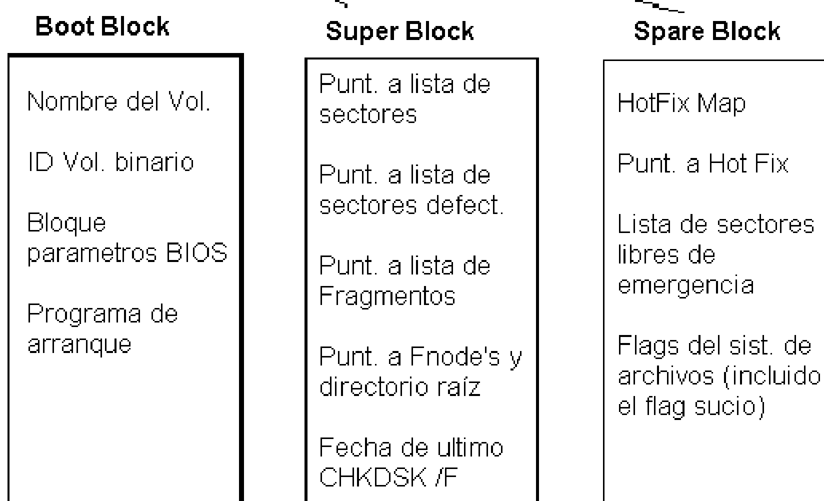


Fig. 15.15



El resto del disco se divide en fragmentos de 8mb. Ver *Figura 15.15*.

Cada fragmento tiene su propio mapa de sectores libres. Como cada bit representa un sector, a este mapa se lo denomina bitmap (mapa de bits) de sectores libres donde cada bit es interpretado así:

- Si el bit = 0: el sector esta ocupado
- Si el bit = 1: el sector esta libre

Este mapa esta situado al principio o al final de cada fragmento, esto permite reservar 16mb de espacio libre contiguo para un mismo archivo.

Hay un fragmento especial localizado en el centro del disco que se denomina bloque de directorio y tiene un tratamiento especial.

15.7.3.3. - **Archivos y Carpetas (Fnode's)**

Cada archivo o carpeta esta asociado a una estructura que se denomina Fnode. Ver *Figura 15.16.*

Cada Fnode ocupa un sector y contiene los siguientes datos del archivo o carpeta:

- Información histórica
- Información de control
- Los atributos
- La lista de control de acceso
- La longitud
- Los primeros 15 caracteres del comienzo del nombre
- La estructura de espacios ocupados

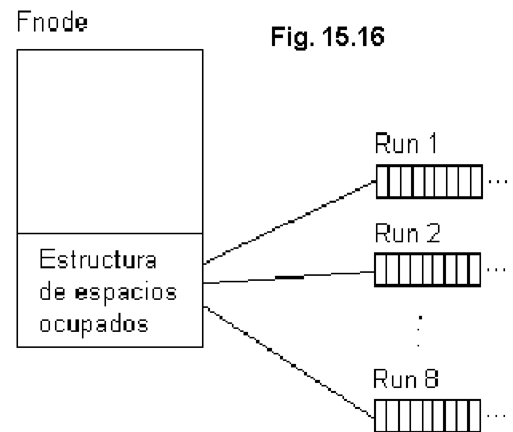


Fig. 15.16

HPFS ve al archivo como una colección de runs[1], donde cada run es una secuencia de sectores contiguos.

Cada estructura de espacios ocupados puede contener hasta ocho punteros a runs distintos de manera que el espacio total de almacenamiento que puede soportar un fnode es de 16Mb.

Si un archivo ocupa mas de 16mb no podrá ser almacenado de manera contigua.

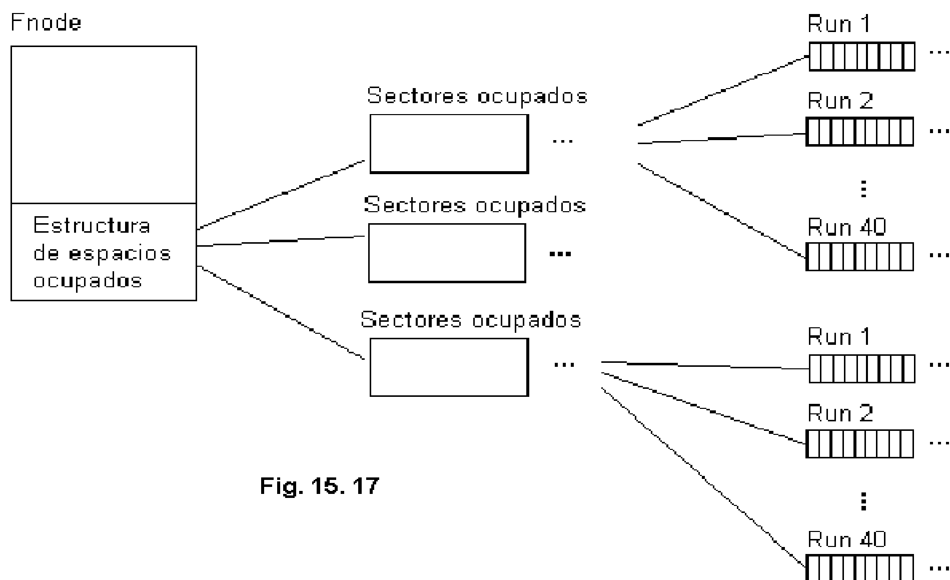


Fig. 15. 17

15.7.3.1. – Estructura.

La estructura del run es la siguiente:

- 32bit: dirección de comienzo del 1er sector
- 32bit: cantidad de sectores

Desde el punto de vista de una aplicación un archivo se ve como una secuencia de bytes.

Para los archivos mayores a 16mb o que se encuentran muy fragmentados HPFS utiliza estructura de espacios ocupados. Esta estructura es básicamente un árbol-b (B-Tree), Ver *15.16.* Mas adelante explicaremos de qué se trata esta estructura de datos.

15.7.3.3.1. - Estructura de un árbol-b en general:

Un árbol-b es un árbol n-ario donde cada nodo puede contener mas de un elemento, en una lista ordenada.

Esta agrupación se utiliza para minimizar la modificación de la estructura en las inserciones y acelerar los accesos a los datos.

Si la agrupación es mínima (un elemento), el árbol-b es un árbol n-ario, y no se aprovecha la particularidad de esta estructura.

[1] El run de HPFS es conceptualmente similar al cluster de DOS: una secuencia de sectores contiguos.

Si la agrupación es máxima, (n elementos), el árbol-b pasa a ser una secuencia y por lo tanto se pierden las propiedades de árbol, o sea es como trabajar con una secuencia.

A la cantidad de elementos que se pueden almacenar como máximo en un nodo se lo denomina orden. *Orden* del árbol-b.

Para un Sistema de archivos, lo mas apropiado es que el *orden* del árbol-b sea múltiplo del tamaño del sector, o del run, en el caso de HPFS.

En cierta forma un árbol-b es una extensión de los árboles 2-3, los cuales se diferencian de los árboles comunes en que no tienen una cantidad de hijos por nodo estricta. Esta pequeña libertad tiene grandes implicancias al utilizar la estructura sobre un disco. Dado que muchas hojas se encuentran parcialmente llenas, es altamente probable que al insertar un nuevo elemento no haga falta generar un nuevo nodo, y aún si esto último fuese necesario, es probable que no haya que generar un nuevo nodo como padre de las hojas, etc. El resultado neto es la *minimización de los nodos modificados* al efectuar una inserción, a costa de una mayor ineficiencia en el aprovechamiento del espacio.

Los árboles B se diferencian de los 2-3 solamente en la cantidad de hijos que pueden tener. Mientras que un árbol 2-3 puede tener 2 o 3 hijos (si no es hoja, claro está) un árbol B de orden n puede tener entre n/2 y n hijos. Esto permite *adaptar el orden del árbol según el tamaño de nodo que se desee obtener*, característica importante si se desea utilizar exactamente un sector por nodo, como se comentó anteriormente.

Otra diferencia importante de un árbol-b respecto a un árbol n-ario es que los datos se almacenan en las hojas mientras que en los nodos internos solo se almacenan referencias a los otros nodos.
En los árboles n-arios cada nodo tiene un dato, y todos los nodos (hojas y nodos internos) son iguales.

Esta estructura garantiza las siguientes características:

- Inserción en $O(\log_{orden} n)$
- Búsqueda en $O(\log_{orden} n)$
- No depende de la “buena” distribución de los datos para balancearse.
- No requiere estar totalmente en memoria para garantizar estos ordenes (una cache acompaña casi siempre a esta estructura pues no siempre es conveniente ni se puede cargar todo el árbol al mismo tiempo).

15.7.3.4. - **Directorios**

Los directorios también están representados por los Fnode’s y en el SuperBlock se encuentra en puntero al directorio raíz.

Los directorios están almacenados en bloques de 2k (cuatro sectores consecutivos) y pueden crecer mientras haya espacio disponible.

Normalmente se almacenan en el fragmento de directorio, que esta cerca del centro del disco para accederlo de forma mas eficiente.

Si este fragmento no es suficiente se utiliza el primer fragmento libre disponible.

Cada bloque de directorio tiene una o mas entradas de directorio, y cada entrada de directorio tiene los siguientes campos:

- Sello temporal
- Puntero a Fnode
- Contador de uso (para los programas de mantenimiento y cache)
- Nombre
- Puntero a un árbol-b

Cada bloque comienza con un numero que indica el tamaño de cada entrada, de manera que el tamaño sea aprovechado sin perder la performance.

En promedio los nombres de directorio contienen 13 letras y alrededor de 40 entradas.

Cada bloque esta ordenado por orden alfabético y la ultima entrada se usa para marcar que no hay mas entradas.

Si el bloque no es suficiente para almacenar un directorio, se utiliza un árbol-b para almacenar la información restante. La cantidad de entradas variable de un bloque hace que el árbol-b sea una estructura apropiada para almacenar este tipo de información.

Para ilustrar la performance que se logra con este tipo de estructura tomaremos como ejemplo el siguiente caso:

> Supongamos que tenemos 40 entradas por directorio, un árbol-b de dos niveles podrá almacenar 1640 entradas y uno de 3 niveles, 65.640.

Un archivo particular podrá ser encontrado en un directorio de 65.640 entradas como mucho con 3 accesos al disco.

Bajo las mismas hipótesis, en una FAT de DOS este mismo acceso provocara 4000 accesos al disco en el peor caso.

Uno de los problemas que presenta esta estructura es cuando se necesita agrandar el tamaño de una entrada, por ejemplo, cuando se hace un rename[2] de un archivo.

HPFS soluciona esto manteniendo un pequeño pool de bloque libres en el directorio de emergencia. (El puntero a este bloque se encuentra en el SpareBlock[3]).

15.7.3.5. - **Manejo de errores**

El mecanismo principal para el manejo de errores se llama hotfix (arreglo rápido). El puntero al bloque hotfix se encuentra en el SpareBlock.

Cuando se detecta un sector defectuoso, se toma un sector del pool de sectores del hotfix para remplazar el sector dañado.

El bloque hotfix contiene una secuencia de doble words (32bit x 2) que representan:

- 32bit: Nro. de sector defectuoso
- 32bit: Nro. se sector que lo reemplaza dentro el hotfix

La cache del sistema de archivos amortigua toda la posible perdida de performance que puede causar este acceso indirecto a estos sectores.

Una de las tareas del CHKDSK[4] es vaciar el bloque hotfix.

Por cada reemplazo que existe en el hotfix, CHKDSK, buscara un sector libre (y bueno) dentro de los fragmentos libres regulares donde recolocar el sector originalmente dañado, actualizando los mapas de archivos que sean necesarios.

En otras palabras el bloque hotfix es un área de almacenamiento temporal de sectores dañados.

15.7.3.5.1 - El "bit sucio"

Existen otros tipos de fallas que no pueden ser previstas por un sistema de archivos, algunas de ellas son:

- Fallas de energía
- Fallas causadas por Virus
- Reseteo accidental del equipo (no se actualiza la FAT del disco con respecto a la FAT que esta en memoria, en especial los datos que se encuentran en cache)

Por este motivo, en el SpareBloque se mantiene el DirtyFS (flag sucio).

Este bit es apagado cuando el sistema se cierra normalmente y esta consistente.

Cuando el sistema arranca, se verifica este flag y si esta prendido se ejecuta el CHKDSK, de manera de reconstruir las partes dañadas que sean necesarias.

Para facilitar el trabajo de reconstrucción y aumentar la robustez del sistema, HPFS mantiene información redundante de las partes criticas, por ejemplo utiliza listas doblemente encadenadas en lugar de listas regulares y también almacena en cada Fnode el principio del nombre del archivo o directorio que representa.

15.7.3.6. - **Atributos extendidos**

HPFS cuenta con un sistema de atributos dinámicos para cada archivo o directorio (EAs[5]).

En lugar de mantener una cantidad fija de bits estática asociada con cada archivo, se puede mantener un bloque dinámico de hasta 64k de información relacionada con los atributos del archivo.

Si se necesita mas que eso, se almacenaran los atributos en un árbol-b fuera del Fnode.

Los EAs mantienen los atributos convencionales de la FAT (Solo lectura, sistema, oculto, archivo) además de otros atributos respecto al tipo de archivo, que no mantiene la FAT. Por ejemplo el tipo de archivo y los permisos sobre el.

Esto posibilita además almacenar para cada archivo una lista de control de acceso (LCA) en caso de que el sistema operativo soporte esta funcionalidad.

15.7.3.7. - **Sistema de archivos instalable**

HPFS trae soporte para leer volúmenes de diferente estructura de otros sistemas operativos.

Para los volúmenes que no son HPFS se puede instalar en el sistema operativo el driver (controlador) que interpreta ese volumen de manera a las aplicaciones les resulte transparente acceder a volúmenes de diferente tipo.

[2] "rename" es una abreviatura del comando que se usa para modificar el nombre de un archivo o directorio.
[3] Spare en ingles significa repuesto.
[4] CHKDSK es la abreviatura de Check Disk, un utilitario de mantenimiento del disco. En Windows y Dos este utilitario lleva el mismo nombre.
[5] EAs es la abreviatura de Extended Atributes que en castellano significa atributos extendidos.

15.7.3.8. - **Comparación entre FAT y HPFS:**

A continuación se presentan las diferencias principales entre DOS y HPFS:

	FAT	HPFS
Longitud máxima de un nombre de archivo o directorio	11 en formato 8.3	254
Nro. de puntos (.) máxima permitida en un nombre	1	muchos
Atributos de un archivo	Bitmap estático	Bitmap estático mas 64k de datos dinámicos
Longitud máxima de una ruta	64	260
Mínimo espacio requerido para representar un archivo (sin contar los datos en si)	Una entrada de directorio (32 bits)	Una entrada (largo variable) + 512 bytes para el Fnode
Espacio promedio desperdiciado por archivo	½ cluster (típicamente 2048 bytes o mas)	½ sector (256 bytes)
Mínima unidad de almacenamiento	Cluster (típicamente 4096 bytes o mas)	Sector (512 bytes)
Espacio ocupado para la FAT (o mapa de archivos)	Centralizado en el track central del disco	Localizado cerca del Fnode de cada archivo
Información del espacio libre	Centralizado en el track central del disco	Localizado cerca de los bitmap's de espacio libre
Espacio libre representado por byte	2048 bytes (½ cluster con 8 sectores por cluster)	4096 bytes (8 sectores)
Estructura de directorio	Lista encadenada, búsqueda secuencial	Árbol-b, búsqueda binaria y agrupada.
Ubicación del directorio	Directorio raíz en el track central, el resto esparcido	Localizado cerca del volumen central
Estrategia de reemplazo de cache	LRU	LRU modificada, sensitiva al tipo de datos y datos históricos
Lectura adelantada	No hasta DOS4. El DOS4 y posteriores implementan una versión primitiva de esta técnica	Lectura adelantada sensitiva al tipo de datos y datos históricos
Escritura en segundo plano	No	Si

15.7.3.9. - **Resumen**

HPFS resuelve todos los históricos problemas del sistema FAT. Alcanza un excelente desempeño para el peor caso tanto para muchos archivos chicos como para algunos archivos grandes gracias a sus estructuras de datos avanzadas y a sus técnicas de lectura adelantada y de escritura en segundo plano.

El espacio en disco se utiliza en forma económica porque se puede "direccionar" a sector.

Las aplicaciones viejas que trabajan con nombre de archivos cortos son compatibles con HPFS y pueden modificarse fácilmente para aprovechar las ventajas de los "nombres largos".

CAPITULO 16

PLANIFICACION DE LA CARGA

16.1 - Introducción

Los objetivos de la Planificación de la Carga son :
- La ejecución de la mayor cantidad de "trabajos" en el menor tiempo posible.
- La no saturación de los recursos.

16.2. - SISTEMAS BATCH.

En los sistemas batch es donde existe una real planificación de la carga, ya que frente a un conjunto de trabajos posibles a ser ejecutados, es posible realizar la selección de los mismos de acuerdo a algún criterio.
A continuación detallaremos los más usuales.

16.2.1. - Tiempo de llegada o planificación secuencial.

A medida que llegan los trabajos van ingresando al sistema hasta la saturación de algún recurso (por ejemplo, que no haya más particiones disponibles).

16.2.2. - El más corto primero.

Ante la posibilidad de seleccionar entre varios trabajos, se seleccionan los más cortos, para que **menos** programas terminen más tarde.
Esta selección es sólo posible frente a un conocimiento previo del tiempo que necesitará cada trabajo.

16.2.3. - Demoras

Para poder determinar entre dos planificaciones distintas cuál es la mejor se introduce en elemento de medida que definiremos así:

Demora Ponderada = Demora Absoluta / Duración

definiendo Demora Absoluta el tiempo desde que un trabajo ingresa y espera ser ejecutado hasta que termina su ejecución y Duración al tiempo de ejecución de un trabajo.

Si tomamos el siguiente ejemplo:

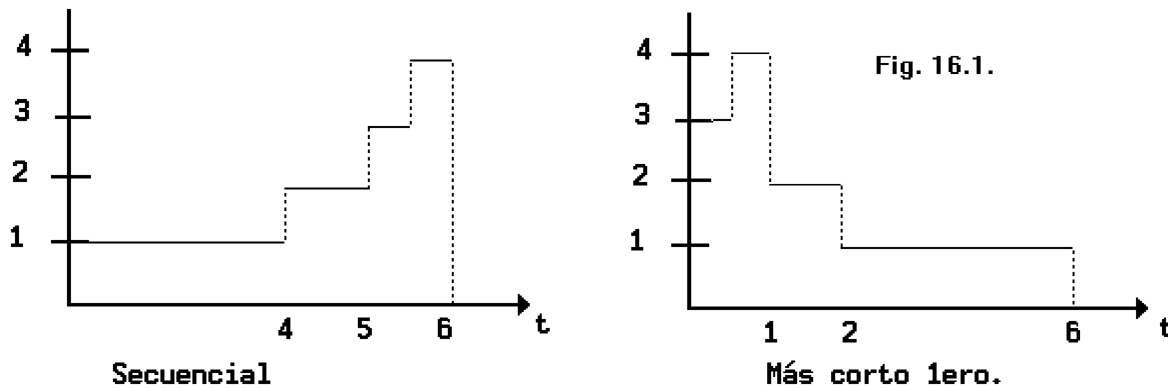


Fig. 16.1.

Trabajo	Duración
1	4
2	1
3	0,5
4	0,5

y los ejecutamos en monoprogramación (Fig. 16.1) en los dos sistemas de Planificación recién vistos resulta que en ambas planificaciones se termina todo en el mismo tiempo, pero será alguna más eficiente que la otra ?.

Apliquemos el elemento de medida.

Secuencial

Trab	Duración	Dem. Absoluta	Dem. Ponderada
1	4	4	1
2	1	5	5

3	0,5	5,5	11
4	0,5	6	12
Promedios		5.125	7,25

El más corto primero

Trab	Duración	Dem. Absoluta	Dem. Ponderada
3	0,5	0,5	1
4	0,5	1	2
2	1	2	2
1	4	6	1.5
Promedios		2,38	1,625

Si observamos la Demora Ponderada en ambos casos vemos que es mucho menor en el más corto primero. Lo que indica que tendremos más usuarios "satisfechos" antes, si bien en ambos casos el tiempo total de ejecución es el mismo.

La Demora Ponderada es un buen índice de medición, ya que independiza a los trabajos de los tiempos de su propia duración

16.2.4. - **Planificación con conocimiento futuro**.

Se asemeja al más corto primero.

Dado un trabajo largo por ingresar, pero conociendo que en poco tiempo ingresarán otros cortos, es posible que convenga esperar la ejecución de esos trabajos cortos, según se ve en el ejemplo:

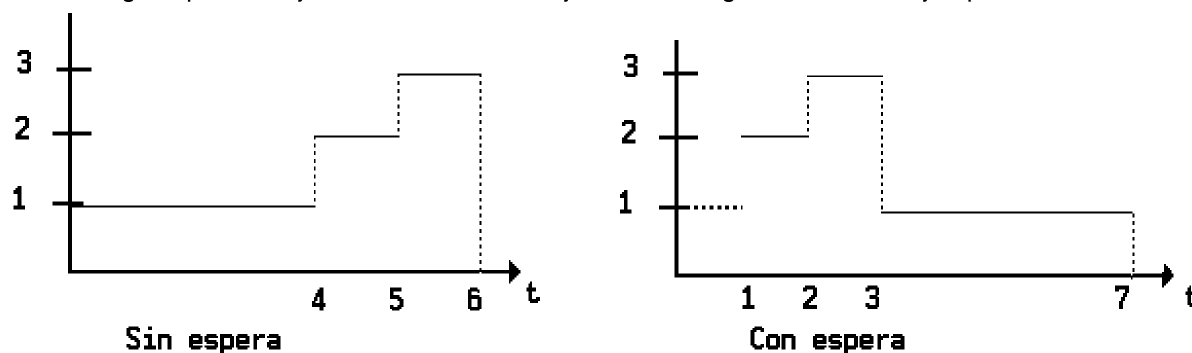


Fig. 16.2.

Sin espera

Trab	Duración	Dem. Absoluta	Dem. Ponderada
1	4	4	1
2	1	5	5
3	1	6	6
Promedios		5	4

Con espera

Trab	Duración	Dem. Absoluta	Dem. Ponderada
1	4	7	1.75
2	1	2	2
3	1	3	3
Promedios		4	2.25

16.2.5. - **Planificación por Mejor Aprovechamiento de los Recursos**.

Veamos el siguiente ejemplo (100K de memoria, 4 unidades de cinta y multiprogramación).

Nota: Consideraremos que la existencia de multiprogramación no afecta la duración de los trabajos, esto no es lógicamente así pero lo utilizaremos para simplificar los cálculos.

Trab	Duración	Memoria	Cintas
1	4	50 K	1
2	1	50 K	2
3	0,5	50 K	2
4	0,5	50 K	3

Planificación Simple

Trab	Duración	Dem. Absoluta	Dem. Ponderada
1	4	4	1
2	1	1	1

3	0,5	1,5	3
4	0,5	2	4
Promedios		2.13	2.25

Más corto primero

Trab	Duración	Dem. Absoluta	Dem. Ponderada
1	4	5	1,25
2	1	2	2
3	0,5	0,5	1
4	0,5	1	2
Promedios		2,375	1,5625

Mejor uso de Cintas

Trab	Duración	Dem. Absoluta	Dem. Ponderada
1	4	4,5	1,125
2	1	1	1
3	0,5	0,5	1
4	0,5	1,5	3
Promedios		1,875	1,53

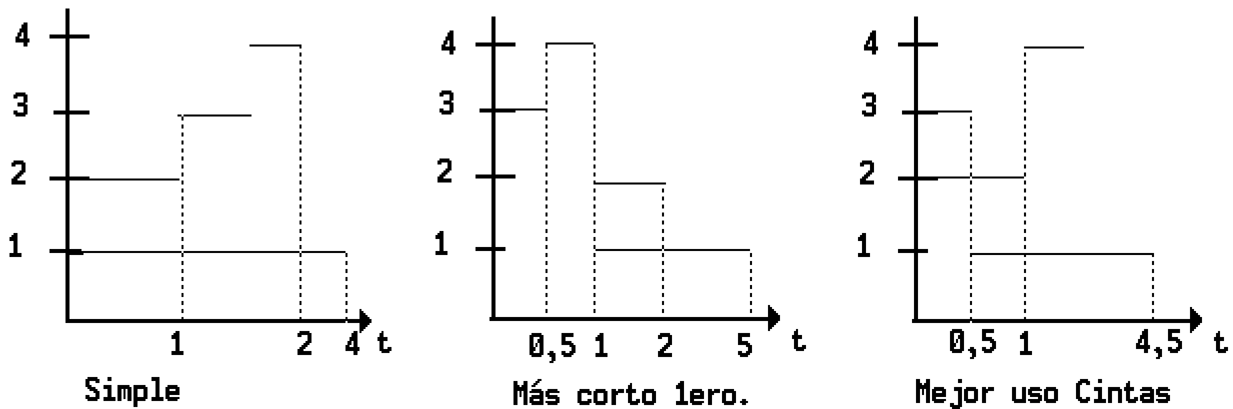


Fig. 16.3.

El mismo criterio puede aplicarse para cualquier otro recurso o combinación de ellos, como memoria, dispositivos, uso de procesador, etc.

16.2.6. - **Planificación por Agotamiento de Recursos**.

Esta planificación permite la entrada de trabajos hasta que un recurso o grupo de recursos se saturan, por lo tanto puede ser por:

- cantidad de memoria,
- que la paginación no exceda un determinado valor,
- hasta el 100% de uso del procesador (en este caso es necesario verificar que el procesador sea en realidad utilizado por procesos usuario y no por el sistema operativo, en general si éste utiliza más del 5% del tiempo de procesador existe algún otro recurso utilizado en exceso, por ejemplo alta paginación).
- que la operaciones de E/S no superen un determinado nivel (por ejemplo, en general una utilización del canal mayor al 30% es excesiva).

16.2.7. - **Planificación por prioridades**.

Esta planificación provoca que se rompan todos los esquemas anteriores de mejor uso de recursos. Se utiliza cuando un determinado trabajo debe ser ejecutado, no importando lo que se está ejecutando en ese momento.

Es el caso, no frecuente, en que se convive con procesos Industriales de Control, los cuales deben ser ejecutados bajo pena de destrucción física o la necesidad de obtener resultados determinados para una hora determinada del día para tomar decisiones financieras, etc.

16.2.8. - **Planificación Algorítmica**.

Se elige un algoritmo que represente mejor los puntos débiles del sistema y con él se seleccionan los trabajos.

Un ejemplo de algoritmo es el de Balance entre uso de procesador y operaciones de Entrada/Salida.

Se solicitan los tiempos estimados de procesador (TCPU) y cantidad de operaciones estimadas de E/S (OPES) con los cuales se realizan las siguientes operaciones:

TOPES = OPES * (T.Posicionamiento + T.Latencia + T.Transferencia)
Duración Teórica = TOPES + TCPU
y se obtiene un coeficiente :
Coef = TCPU / Duración Teórica que es menor o igual a 1

Por ejemplo, el coeficiente en $TCPU_1$ / ($TCPU_1$ + $TOPES_1$) + $TCPU_2$ / ($TCPU_2$ + $TOPES_2$)
será = 1 cuando
$TOPES_1 = TCPU_2$
Y $TOPES_2 = TCPU_1$
Esto implica que las ráfagas de CPU de un proceso coinciden con las ráfagas de E/S del otro proceso.
En el caso de tres procesos la fórmula sería igual a 1 cuando:
$TOPES_1 = TCPU_2 + TCPU_3$
$TOPES_2 = TCPU_1 + TCPU_3$
$TOPES_3 = TCPU_1 + TCPU_2$

Luego se permitirá entrar la cantidad de trabajos que cumplan :
$\sum \text{Coef} \leq 1$

16.2.9. - **Planificación por Balance**.

Se busca cargar una mezcla de trabajos de mucha E/S con otros de alto uso de procesador.
Esto generalmente se utiliza con una planificación similar a la de prioridades en la cual se ejecutan durante el día trabajos cortos y de poco uso de procesador y se deja para la noche los de alto uso de procesador y poca E/S.

16.3. - **SISTEMAS INTERACTIVOS**.

Aquí no es posible establecer una planificación de antemano, solo se puede intentar que los tiempos de respuesta a los usuarios sean razonables.
Luego su objetivo es mantener satisfechos al usuario que está delante de un puesto de trabajo, y generalmente se utiliza el método de darle prioridad al usuario altamente interactivo.

16.3.1. - **Planificación por Contención**.

Significa admitir hasta **n** usuarios, el usuario **n+1** es rechazado. Si existiese alguno de mayor prioridad que desea entrar, se debería forzar a alguno de baja prioridad.
También en este caso puede utilizarse alguna de las variantes de agotamiento de recursos.

16.3.2. - **Planificación Ponderada**.

Cada tipo de usuario tiene asociada una carga, por ejemplo :

Común interactivo	1
Batch	1,5
Priorizado	0,5

Se determina cuantas unidades es capaz de soportar el sistema, por ejemplo 50, y pasadas estas unidades no se aceptan más usuarios.
Los usuarios son agrupados y cada grupo tiene un máximo de unidades, cuando es excedido ese máximo no se aceptan más usuarios del grupo.

16.3.3. - **Planificación Algorítmica**.

Esta se basa en que la utilización de algún recurso debe ser, en general, menor a un valor tope.
Utilización (recursos) = Tiempo Uso Recurso / Tiempo de Observación
< Tope >
Contención (recurso) = Procesos en espera / Tiempo de Observación
< Tope > (ideal 0)
Servicio (t. de respuesta prom.) = (Tiempo Servicio + Tiempo Cola) / Tiempo Observación
< Tope >
Proporción = (CPU (S.O.) + CPU.(Usuarios)) / CPU (Usuarios)
(lo más cercano posible a 1)
Para saber si el sistema anda bien, o se puede permitir el ingreso de un nuevo usuario, se debe verificar no exceder los valores máximos indicados (Topes) que correspondan a un sistema en particular.

TEORIA DE COLAS

16.4. **Modelización estocástica de los instantes de llegada y duración de trabajos**

16.4.1. **Introducción**

Consideraremos el caso en que **no** tenemos un sistema batch. Para eso vamos a estudiar la situación general en la que los trabajos llegan en instantes aleatorios para ser atendidos y cada uno necesita un tiempo de atención que es aleatorio y que no conocemos con anticipación. No podremos hacer una planificación de la carga pero, bajo ciertos supuestos bastante razonables, vamos a obtener conclusiones útiles sobre el comportamiento del sistema. Los resultados que vamos a ver son parte de una rama de la teoría de probabilidades llamada *teoría de colas*.

16.4.2. **Descripción del modelo y notación**

Los trabajos llegan en instantes aleatorios para ser procesados. A pesar de que no sabemos en qué instante va a llegar un trabajo, existe un promedio o **tasa de arribos** por unidad de tiempo, que denotamos con la letra λ. Se supone que si observamos el sistema durante un tiempo no muy corto, podemos contar el número de arribos y dividiéndolo por el tiempo transcurrido vamos a obtener un buen estimador de λ.

Por ejemplo: Observamos una ventanilla y su cola durante 3 horas. En ese período llegaron 45 personas para hacer un trámite allí. Cuánto vale el estimador de λ?. En qué unidades?. Por ejemplo, en este caso λ sería 45 personas/hora o 0,25 personas/minuto.

c será el número de puntos de atención (**despachadores**) para los trabajos. Es el número de procesadores (o de cajeros cuando se modeliza la atención a clientes en un banco, por ejemplo) que están disponibles para ocuparse con las tareas que van llegando. Cuando llega un trabajo, si hay un despachador libre, es atendido por éste hasta su finalización. Si no, espera en una cola. Cuando se libera algún despachador, si hay cola, alguno de la cola va inmediatamente a ser atendido.

μ denotará la **tasa de atención**, común a los *c* despachadores. Es el número de trabajos que un despachador es capaz de atender por unidad de tiempo. Lo podemos estimar contando la cantidad de atenciones por parte de un despachador en un período no muy corto y dividiendo por el tiempo transcurrido menos el tiempo en que el despachador estuvo ocioso. Si hay varios despachadores se realiza esa estimación para cada uno y luego se promedia (recordar que suponemos que todos atienden con la misma tasa).

Supondremos que $\lambda < c\,\mu$ porque de no ser así, estaríamos incluyendo casos en los que se incrementa indefinidamente la cantidad de trabajos por atender sin posibilidad de retorno.

Por ejemplo en el caso de caso $\lambda = 15$ personas/hora y teniendo 4 despachadores μ podría ser 4, es decir si cada despachador atiende a 4 personas/hora podemos satisfacer la necesidad del sistema ya que

$$15 < 4 * 4 = 16$$

Diremos que el sistema se encuentra en estado *i* (con $0 \leq i < \infty$) si el número de trabajos en el sistema, contando los que se están atendiendo y también los que esperan, es *i*.

p_i será la **probabilidad de que en un instante cualquiera el sistema se encuentre en estado *i***. Una interpretación práctica de esta magnitud esta dada por la proporción de tiempo (a largo plazo) en que el sistema tiene *i* trabajos.

Por ejemplo: Si observo al sistema durante 3 horas y veo que la suma de los períodos en que hubo 3 personas en el sistema es de 25 minutos entonces P_3 será aproximadamente 25/180 = 0,138.

Sabemos que la $\sum p_i = 1$ con i de 0 a ∞ es igual a 1 por ser probabilidades, entonces podrían tenerse observando el sistema digamos por un lapso de 5 horas, probabilidades del estilo de :

$P_0 = 20 / 300$
$P_1 = 30 / 300$
$P_2 = 50 / 300$
$P_3 = 0$
$P_4 = 0$
$P_5 = 200 / 300$
$P_6 = p_7 = p_8 = 0$

De todas maneras aspiramos a *calcular* las p_i a partir de λ, μ, c y los supuestos del modelo. Así podremos, por ejemplo, ver cómo se modifica la proporción de tiempo en que el sistema está ocioso (p_0) cuando duplicamos el número de despachadores, o qué efecto tiene una modificación en la velocidad de atención (μ) en la probabilidad de que haya cola ($\Sigma_{i>c}\, p_i$).

16.4.2.1. **Más supuestos sobre el modelo. Ecuaciones de balance.**

Los tiempos entre llegadas y los tiempos de atención son independientes entre si, en el sentido de que, por ejemplo, un tiempo de atención no da información sobre lo que durará la atención siguiente. Sí sabemos, como se dijo, que se verifican las frecuencias medias λ y μ. Los tiempos también son independientes del estado en que se encuentra el sistema. Esto implica, entre otras cosas, que con cola FIFO o LIFO obtendremos los mismos resultados sobre las p_i porque da lo mismo cuál trabajo tomemos de la cola para dárselo a un despachador. Existen variantes de este modelo que no suponen independencia.

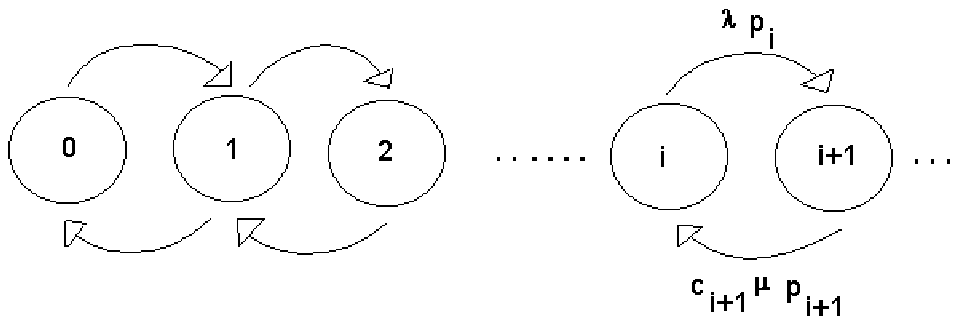
Finalmente suponemos que para cada estado *i* se cumple la *ecuación de balance*

$$p_i\, \lambda = p_{i+1}\, c_{i+1}\, \mu$$

donde $c_k = k$ si $k < c$, y $c_k = c$ si $k \geq c$.

c_i es la cantidad de despachadores ocupados cuando el sistema está en el estado i.

El primer miembro de la igualdad es la tasa de pasaje del estado i al estado i+1, en tanto que el segundo miembro de la igualdad es la tasa de pasaje del estado i+1 al estado i.



Estas ecuaciones pueden deducirse de supuestos más elementales, dentro de la teoría de procesos de Markov. Estos consisten en considerar que los tiempos entre arribos y de atención siguen distribuciones exponenciales. Se puede ver además que la distribución exponencial describe muy bien el tipo de aleatoriedad que estamos manejando. Aquí vamos a prescindir de esas deducciones de las ecuaciones de balance viendo que ellas mismas constituyen suposiciones razonables.

En efecto, dado que p_i es la proporción de tiempo en que hay *i* trabajos en el sistema y λ es la cantidad de arribos por unidad de tiempo, tenemos que $p_i\, \lambda$ representa la tasa de salida del estado *i* hacia el estado *i+1*. Por otro lado, como c_{i+1} es la cantidad de despachadores atendiendo cuando hay *i+1* trabajos en el sistema, $p_{i+1}\, c_{i+1}\, \mu$ representa la tasa de salida del estado *i+1* hacia el estado *i*. Las ecuaciones de balance dicen entonces que a largo plazo (digamos en tiempo infinito) se iguala la cantidad de pasajes de *i* a *i+1* con los pasajes de *i+1* a *i*. Esto se debe a que si se produce un pasaje entre estos dos estados en un sentido, el siguiente pasaje que involucre estos dos estados deberá ser en el sentido opuesto. Así que en tiempo infinito los pasajes en un sentido se balancean con los del sentido opuesto, a no ser que a partir de un instante deje de haber pasajes entre estos dos estados (en cuyo caso podría haber hasta ese momento un pasaje más en un sentido que en otro). Vamos a descartar esto último notando que puede ocurrir solamente en dos casos razonablemente imposibles para nuestro modelo:

Primero: El número de trabajos se agrande irreversiblemente y entonces un paso de *i* a *i+1* nunca es compensado por el opuesto.

Segundo: Este sería el caso en que el número de trabajos en el sistema queda, a partir de un instante, siempre más chico que *i* o siempre más grande que *i+1* pero sin aumentar irreversiblemente en tamaño. Entonces deberá existir algún estado distinto de *0* por el que se pasa infinitas veces pero tal que a partir de un cierto instante, siempre que se sale de él, es en un único sentido.

Ambos casos pueden suponerse imposibles en nuestro modelo. Así completamos una justificación heurística de las ecuaciones de balance.

La notación de Kendall (muy difundida) llama *M/M/c* a un sistema como el descripto (M de Markoviano). Esta notación se refiere a que los arribos y las atenciones siguen la distribución exponencial, lo que hace markovianos a los procesos, y a que hay c despachadores. Podemos tener $c = \infty$, en el caso en que consideramos que siempre va a haber suficientes despachadores como para atender todos los trabajos que lleguen, sin que se forme cola. Se ha usado este último modelo para representar un disco que por tener un gran número de brazos móviles, va a poder atender en el acto toda solicitud de disco que se presente.

16.4.3. **Cálculos para un sistema *M/M/1***

En este caso tenemos c = 1, es decir $c_0 = 0$, $c_1 = 1$, $c_2 = 1$, …etc., siempre hay un despachador atendiendo excepto en el caso en que no hay trabajos para atender. Luego en virtud de las ecuaciones de balance:

$$p_0\, \lambda = \mu\, p_1$$

$$p_1 = p_0\, (\lambda / \mu)$$

$$p_2 = p_1\, (\lambda / \mu) = p_0\, (\lambda / \mu)^2$$

y en general, para $i = 1, 2, \ldots$:

$$p_i = p_0\, (\lambda / \mu)^i$$

Como las p_i son probabilidades (o fracciones de un tiempo total), deben sumar 1. Así:

$$1 = \Sigma_{i \geq 0}\; p_i = \Sigma_{i \geq 0}\, p_0\, (\lambda / \mu)^i = p_0\, (1 - (\lambda / \mu)^{-1}$$ (sumando la serie geométrica, que es posible ya que $\lambda / \mu < 1$ por que $\boldsymbol{\lambda < c\, \mu}$)

Recordemos que $\Sigma_{i \geq 0}\, q^i = 1 / (1 - q)$ si $q < 1$

Luego, despejando de aquí p_0 tenemos

$p_0 = 1 - \lambda / \mu$, y reemplazando más arriba,

$$p_i = (1 - (\lambda / \mu))\, (\lambda / \mu)^i.$$

Por ejemplo, calcular la probabilidad de que haya cola aquí sería $\Sigma_{i\; > C}\; p_i = 1 - p_0 - p1$

Llamemos $N_s = N_c + N_a$ a la cantidad (aleatoria) de trabajos que tiene en un instante el sistema. N_a son los que están siendo atendidos y N_c los que están en la cola. La esperanza (o valor promedio) del número de trabajos en el sistema, $E(N_s)$, será la suma de todos los posibles números de trabajos en el sistema, cada uno multiplicado por la probabilidad (o fracción de tiempo) que le corresponde:

$$\Sigma_{i \geq 0}\, i\, p_i = \Sigma_{i \geq 0}\, i\, (1 - (\lambda / \mu))\, (\lambda / \mu)^i = (1 - (\lambda / \mu))\, (\lambda / \mu)\, \Sigma_{i \geq 0}\, i\, (\lambda / \mu)^{i-1}$$

$$= (1 - (\lambda/ \mu))\, (\lambda/\mu)\, (1 - (\lambda/ \mu))^{-2}$$ (sumando la derivada de la serie geométrica)

Entonces, $E(N_s) = (\lambda/\mu)\, (1 - (\lambda/ \mu))^{-1} = ((\mu / \lambda) - 1)^{-1} = \lambda / (\mu - \lambda)$

$E(N_c)$ será una suma análoga a la anterior pero a i trabajos atendiéndose le corresponden $i - 1$ trabajos en la cola si $i > 0$, y 0 trabajos si $i = 0$. Luego:

$$E(N_c) = 0\, .\, p_0 + \Sigma_{i \geq 1}\, (i - 1)\, p_i = \Sigma_{i \geq 1}\, i\, p_i - \Sigma_{i \geq 1}\, p_i = \Sigma_{i \geq 0}\, i\, p_i - (1 - p_0) = E(N_s) - (1 - p_0) = E(N_s) + p_0 - 1.$$

Reemplazando tenemos una expresión alternativa:

$$E(N_c) = (\lambda / \mu)\; (1 - \lambda / \mu)^{-1} + 1 - \lambda / \mu - 1 = (\lambda / \mu)^2 (1 - (\lambda / \mu))^{-1}.$$

Para calcular $E(N_a)$ tenemos en cuenta que hay 0 trabajos atendiéndose si hay 0 trabajos en el sistema, y hay 1 atendiéndose si hay 1 o más en el sistema. Luego:

$$E(N_a) = 0\, .\, p_0 + \Sigma_{i \geq 1}\, 1.\, p_i = 1 - p_0$$

Notar la aditividad de la esperanza: $E(N_c+N_a) = E(N_c) + E(N_a)$, una propiedad que se cumple en general para la suma de dos variables aleatorias.

16.4.4. ***M/M/c* y *M/M/∞***

Los razonamientos y los cálculos son análogos, salvo que aparecen otros tipos de series para sumar en lugar de la geométrica. Con $c = \infty$, la del desarrollo de la función exponencial y con $1 \leq c < \infty$, aparecen series geométricas salvo los primeros términos y las expresiones son un poco más complicadas, pero manejables.

Veamos por ejemplo el cálculo de las p_i en el caso $c = 3$:

$p_1 = p_0 (\lambda / \mu)$, como con $c = 1$, pero la segunda ecuación de balance es $p_1 \lambda = p_2 2 \mu$. Luego:

$p_2 = p_1 \lambda / (2 \mu) = p_0 (1/2) (\lambda/\mu)^2$,

y como la tercera ecuación de balance es $p_2 \lambda = p_3 3 \mu$, tenemos

$p_3 = p_2 \lambda/(3 \mu) = p_0 (1/6)(\lambda/\mu)^3$.

En este punto observemos que con $c = \infty$, obtenemos en general $p_i = p_0 (1 / i!)(\lambda / \mu)^i$,
y usando que $\Sigma_{i \geq 0} p_i = 1$, junto con la igualdad $e^x = \Sigma_{i \geq 0} x^i / i!$, podemos hallar las p_i.

Volviendo al caso $c = 3$, tenemos como cuarta ecuación de balance $p_3 \lambda = p_4 3 \mu$, porque no hay más que tres despachadores. En general, para $i \geq 4$, $p_{i-1} \lambda = p_i 3 \mu$, y por lo tanto

$p_i = p_{i-1} \lambda / (3 \mu) = p_0 (1/6)(1/3)^{i-3}(\lambda / \mu)^3$. Luego

$$1 = \Sigma_{i \geq 0} p_i = p_0 [1+ \lambda/\mu + (1/2) \Sigma_{i \geq 2} (1/3)^{i-2}(\lambda/\mu)^i]$$

$$= p_0 [1+ \lambda/\mu + (1/2) (\lambda/\mu)^2 \Sigma_{i \geq 2} (1/3)^{i-2}(\lambda/\mu)^{i-2}]$$

$$= p_0 [1+ \lambda/\mu + (1/2) (\lambda/\mu)^2 \Sigma_{i \geq 0} (1/3)^{i}(\lambda/\mu)^{i}]$$

$$= p_0 [1+ \lambda/\mu + (1/2) (\lambda/\mu)^2 (1-\lambda/(3 \mu))^{-1}].$$

De aquí se despeja p_0 (obteniendo que $p_0 = e^{(-\lambda/\mu)}$) y con las ecuaciones de más arriba, el resto de las p_i.

16.4.5. - **Tiempos esperados**

Tiempo esperado de atención de 1 trabajo en el caso M/M/1.

Como μ es la cantidad de trabajos atendidos en una unidad de tiempo :

μ	trabajos en	1	unidad de tiempo
1	trabajo en	$1 / \mu$	unidad de tiempo

Ahora, para deducir heurísticamente la esperanza del tiempo que demora un trabajo en la cola y la esperanza del tiempo que demora un trabajo en pasar por todo el sistema, nos ponemos en la posición de un trabajo que llega al mismo

16.4.5.1. - Tiempo esperado de espera en la cola (si la cola es FIFO) - T_c

$E(T_c)$ = "el tiempo para un trabajo multiplicado por la cantidad esperada de trabajos que se tienen que procesar antes de mi llegada al despachador"

$$= 1 / \mu (E (N_s)) = (1 / \mu) \lambda / (\mu - \lambda)$$

$E (N_s)$ es la esperanza de una cantidad de usuarios en el sistema.

16.4.5.2. - Tiempo esperado total en el sistema

Partiendo de que $E(T_s)$ es la esperanza de tiempo de permanencia en el sistema

$E(T_s) = 1/\mu\,(E(N_s)+1)$ "el tiempo para un trabajo multiplicado por la cantidad de trabajos que están antes que yo, más yo mismo (todos estos son atendidos en un tiempo promedio de $1/\mu$ cada uno)"

$$= (1/\mu)\,(\lambda/(\mu-\lambda)+1)$$

$$= (1/\mu)\,\mu/(\mu-\lambda)$$

$$= 1/(\mu-\lambda)$$

16.4.5.3. - Tiempo esperado de atención

Sabemos que $E(T_a) = E(T_s) - E(T_c)$ o equivalentemente $E(T_s) = E(T_c) + E(T_a)$ que viene a ser el tiempo de atención a todos los trabajos incluyéndome menos el tiempo de atención a mí mismo.

Remplazando ahora tenemos

$$E(T_a) = 1/\mu\,(E(N_S)+1) - 1/\mu\,(E(N_S))$$

$$= 1/\mu\,E(N_S) + 1/\mu - 1/\mu\,(E(N_S))$$

$$E(T_a) = 1/\mu$$

Con estos resultados y basándonos en la definición de demora ponderada de un trabajo como cociente entre la demora absoluta y la duración, podemos calcular un cociente entre esperanzas para obtener una magnitud análoga a la demora ponderada pero para un sistema probabilístico M/M/1.

Demora ponderada en el sistema M/M/1 = Demora absoluta / duración

$$= E(T_s)/E(T_a)$$

$$= (1/(\mu-\lambda))\,/\,(1/\mu)$$

$$= \mu/(\mu-\lambda)$$

$$= 1/(1-\lambda/\mu)$$

Ejercicios resueltos

1) *Proponer una manera de estimar μ que involucre los c despachadores.*

Se observa el sistema durante un tiempo. Para cada despachador k ($1 \le k \le c$) se mide el tiempo que no estuvo ocioso, t_k, y el número de personas que atendió, n_k.

n_k / t_k será un estimador de μ que luego se promedia para todos los despachadores.

Es decir: *Estimación de* $\mu = (\Sigma\, n_k / t_k) / c$.

2) *En un sistema M/M/1, la tasa de arribos es de 2.1 por segundo y la de atención de 3.2. Calcular la probabilidad de que el despachador no se encuentre ocioso y la longitud esperada de la cola.*

Probabilidad de que el despachador no esté ocioso = $1 - p_0 = \lambda/\mu = 2.1/3.2 = 0.65625$.

Longitud esperada de la cola = $E(N_c) = (\lambda/\mu)(1-\lambda/\mu)^{-1} - \lambda/\mu = 0.65625\,(.34375)^{-1} - 0.65625$
$= 1.91 - 0.66 = 1.25$, aproximadamente.

3) *Para un sistema M/M/∞, obtener la probabilidad de que el sistema esté ocioso (p_0) en función de la tasa de arribos (λ) y la de atención (μ).*

Como en el apunte, partiendo de las ecuaciones de balance llegamos a que

$p_i = p_0\,(1/i!)(\lambda/\mu)^i$. Luego

$1 = \Sigma_{i \geq 0}\ p_i = p_0\, \Sigma_{i \geq 0}\,(1/i\,!)(\lambda/\mu)^i = p_0\, e^{\lambda/\mu}$. Entonces:

$p_0 = e^{-\lambda/\mu}$.

4) *Para un sistema M/M/1, con tasa de atención 3 veces mayor que la tasa de arribo, hallar la probabilidad de que haya cola.*

Probabilidad de que haya cola $= \Sigma_{i>1}\ p_i = 1 - p_0 - p_1$
$= 1 - (1 - \lambda/\mu) - (1 - \lambda/\mu)(\lambda/\mu) = 1/3 - (2/3)(1/3) = 1/3 - 2/9 = 1/9$.

Algunos libros que incluyen el tema de teoría probabilística de colas

Introducción a los Sistemas Operativos, H. M. Deitel, 2da. Edición, Addison-Wesley.
Sistemas Operativos, S. E. Madnick y J. J. Donovan, Editorial Diana.
Real-Time Systems Design and Analysis. P. Laplante, IEEE Press.

CAPITULO 17

ABRAZO MORTAL - DEADLOCK

17.1. - PROBLEMA

Dos objetos (Procesos) desean hacer uso de un único recurso no compartible o un número finito de ellos, y cada uno de ellos posee la porción que el otro necesita. Un ejemplo clásico es el de dos procesos P(1) y P(2), cada uno de ellos tiene asignadas 2 unidades de cinta y cada uno de ellos necesita una tercera, pero en su sistema de ejecución existen sólo 4 unidades en total. Obviamente estamos en presencia de un deadlock, ya que P(1) espera recursos de P(2) y P(2) espera recursos de P(1).

Para hacer uso de recursos que puedan llevarnos al estado anterior, debería establecerse algún tipo de mecanismo o protocolo:

1. Comenzar a usar el recurso sólo en el caso de que nadie lo esté usando, pues si ya está siendo usado por otro ocurre el "DEADLOCK" (o cualquier otro desastre). Si no se fija si ya se está usando "puede siempre ocurrir" el "DEADLOCK".
2. Si dos procesos desean usarlo en el mismo instante, la solución de arriba no sirve porque o bien los dos comienzan a utilizarlo (deadlock) o ambos esperan que el otro comience, que es otra forma de "DEADLOCK" (starvation - inanición). Esto se puede arreglar por medio de darle prioridad a uno de ellos.
3. Si un grupo de procesos tiene mayor prioridad sobre otro, puede ocurrir que el segundo grupo nunca tome el recurso, luego, hay que inventar un mecanismo que varíe esa prioridad. Por ejemplo, pasado X tiempo, o cuando haya N procesos en espera, la prioridad se invierte.

17.2. - UTILIZACION DE RECURSOS

Dentro de un sistema, con un número de procesos que compiten por recursos no compartibles, se debe tener en cuenta que los recursos son limitados: no se puede pedir más de los que existen y no se puede utilizar un recurso que ya esté siendo utilizado por otro proceso. Luego un proceso puede utilizar un recurso de la siguiente manera:

Pedirlo : (REQUEST) si no puede ser satisfecho esperar. Será incluido en una cola en espera de ese recurso (Tablas).
Usarlo : (USE) el proceso puede operar el recurso.
Liberarlo : (RELEASE) el proceso libera el recurso que fue pedido y asignado.

17.3. - **DEFINICION DE DEADLOCK**

Un conjunto de procesos está en estado de "DEADLOCK" cuando cada proceso del conjunto está esperando por un evento que solo puede ser causado por otro proceso que está dentro de ese conjunto. En el ejemplo de las cintas, el evento que se espera es "liberación de la cinta" (igual tipo de recurso).

También es posible estar frente a casos de "DEADLOCK" de distintos tipos de recursos, por ejemplo si un proceso P(1) tiene asignada una lectora de tarjetas y espera por la impresora, y un proceso P(2) tiene asignada la impresora y espera por la lectora de tarjetas.

17.4. - **CONDICIONES NECESARIAS PARA EL "DEADLOCK"**

Se llega al "DEADLOCK" sii las siguientes 4 condiciones se cumplen **simultáneamente**:

17.4.1 - **Exclusión Mutua**

Al menos un recurso debe ser usado en forma exclusiva (no se puede compartir). Si un proceso lo desea, pero ya lo tiene otro, deberá esperar hasta que sea liberado.

17.4.2 - **Espera y Retenido (Hold & Wait)**

Debe existir un proceso que tiene retenido un recurso y está esperando por otro que a su vez está siendo ocupado por otro proceso.

17.4.3 - **Sin Desalojo**

Los recursos no pueden ser desalojados, es decir que un recurso es liberado en forma voluntaria por un proceso y luego que haya finalizado su tarea.

17.4.4 - **Espera circular**

Debe existir un conjunto de procesos {P(0), P(1),...,P(n)} tal que P(0) espera un recurso ocupado por P(1), P(1) espera un recurso ocupado por P(2), y P(n) espera un recurso ocupado por P(0).

De alguna manera esta última condición implica la de hold & wait.

17.5. - **GRAFO DE ASIGNACIÓN DE RECURSOS**

Estos grafos se utilizan para describir los "DEADLOCKS". Sea G = (V,E), donde V es el conjunto de vértices y E es el conjunto de flechas (arcos orientados).

A su vez V está compuesto por 2 tipos:

P = {p(1),p(2),...,p(n)} conjunto de Procesos

R = {r(1),r(2),......,r(m)} conjunto de Recursos

Los elementos de E son:

(p(i),r(j)) ======> p(i) está esperando una instancia del recurso r(j).

(r(j),p(i)) ======> r(j) (una instancia de) está asignada al proceso p(i).

Si tenemos (p(1),r(3)) y r(3) es asignado a p(1), instantáneamente esto se transforma en (r(3),p(1)) y cuando el recurso es liberado se elimina la flecha.

Ejemplo :

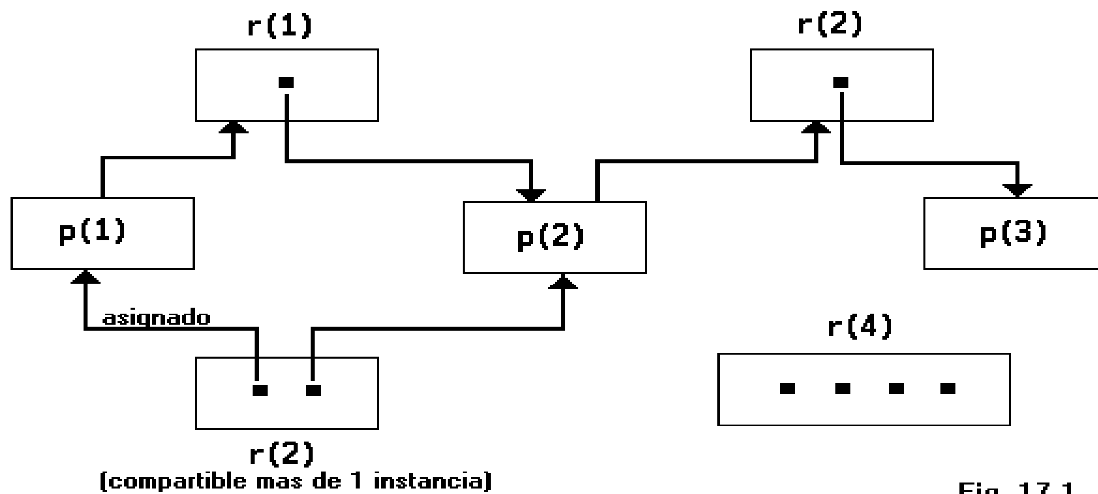


Fig. 17.1.

G { V { P = { p(1), p(2), p(3) }
R = { r(1), r(2), r(3), r(4) } }
E = { (p(1),r(1)), (p(2),r(3)), (r(1),p(2)),
(r(2),p(2)), (r(2),p(1)), (r(3),p(3)) } }

A partir de aquí es fácil ver que si los grafos no forman un ciclo cerrado no hay procesos en **deadlock**. Si existe ciclo cerrado entonces puede existir un **deadlock**.

Si cada tipo de recurso tiene exactamente una sola instancia y tenemos ciclo cerrado entonces estamos en presencia de un deadlock. En este caso un ciclo cerrado es condición necesaria y suficiente para que exista deadlock.

En caso de tipos de recursos con más de una instancia, la existencia de un ciclo cerrado es condición necesaria, pero no suficiente.

EJEMPLO (Fig. 17.2)

Si a nuestro ejemplo agregamos (p(3),r(2)), tendríamos :

p(1) ---> r(1) ---> p(2) ---> r(3) ---> p(3) - --> r(2) --- > p(1)

p(2) ---> r(3) ---> p(3) ---> r(2) ---> p(2)

en el cual se ve claramente que estos dos ciclos cerrados están señalando 2 deadlocks.

Ahora consideremos el ejemplo de la Fig. 17.2 donde tenemos el ciclo cerrado

p(1) --> r(1) --> p(3) --> r(2) -- > p(1)

pero se ve que si p(4) o p(2) liberan sus recursos se rompe el ciclo cerrado, luego no estamos en presencia de un **deadlock.**

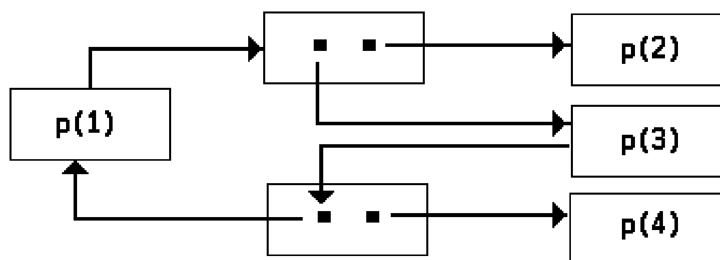


Fig. 17.2.

Al liberar p(2) o p(4) queda liberada una instancia que permite la asignación a alguno de los procesos en espera.

CONCLUSION: *Cuando no existe un ciclo no hay deadlock. Ahora si existe un ciclo puede o no haber deadlock.*

17.6. - **MANEJO DE DEADLOCK**

Existen cuatro maneras de manejar los **Deadlocks** :
- Ignorar su ocurrencia
- No permitir que ocurran (Evitarlo).
- Prevenir que ocurran (Prevención)
- Permitirlo y luego recuperar (es caro y dificultoso).

17.6.1 - **PREVENCION.**

Antes vimos cuáles eran las **condiciones necesarias** para que ocurra un "Deadlock", luego si aseguramos que una de ellas no ocurre, estamos seguros de que no tendremos un deadlock.

17.6.1.1 - **EXCLUSION MUTUA**

Para evitarlo se deberían utilizar siempre recursos que pudiesen ser compartidos, ya que los procesos no deben esperar la exclusividad para usar recursos compartibles, por ejemplo los archivos en modo lectura. Pero esto no es real, pues siempre se necesitarán por ejemplo archivos en modo escritura, cintas, etc. (algunos recursos son intrínsecamente no compartibles)

17.6.1.2 - **ESPERA Y RETENIDO (Hold & Wait)**

Existen dos formas de evitarlo :
1. Un proceso debe tener asignados todos sus recursos antes de comenzar su ejecución.
Si necesita 3 unidades de cinta, no se le asignan las 3 hasta que estén todas disponibles, y el proceso no comienza hasta tanto no se tengan las 3, y las tendrá todo el tiempo en que ese proceso ejecute, aún si las usa solo al final.
2. Si un proceso tiene asignado un recurso no puede asignársele otro hasta que haya liberado al anterior.
Si un proceso que imprime un listado además utiliza un periférico de entrada dedicado y un archivo, cuando termina la utilización del periférico dedicado y el archivo se le desasignan ambos y se le asigna nuevamente el archivo y la impresora hasta su finalización.

PROBLEMAS : Un recurso que es asignado al comienzo del proceso pero solo es utilizado al final de ese proceso es inutilizado mucho tiempo (bajo uso de recurso). O un proceso puede esperar indefinidamente un recurso muy popular. (*Starvation- Inanición*).

17.6.1.3 - **SIN DESALOJO**

Una forma de evitarlo es si un proceso tiene recursos y pide otro no disponible, pierde todos los recursos y solo recomenzará cuando los tenga todos a su disposición.

Otra forma sería si un proceso pide algún recurso, si este está disponible se lo asigna, si no, se controla si ese recurso está asignado a un proceso en espera de algún otro recurso y si es así se lo quita y asigna al solicitante; si no está disponible de ninguna manera, espera.

Esto es lo que se usa en recursos cuyos estados son fácilmente salvables, como memoria y procesador (donde es necesario previamente salvar PSW, registros, etc.).

Piense Ud. cuán engorroso sería en el caso de una impresora (las líneas saldrían mezcladas).

17.6.1.4 - **ESPERA CIRCULAR**

Para asegurar que esta condición no se cumpla consideremos lo siguiente:
A cada "tipo de recurso" le asignamos un número Natural único
R = {r(1),r(2),....,r(n)}   F:R --> N
F(lectora)=1 , F(disco)=5 , F(cinta)=7 , F(impresora)=12
y consideremos el siguiente protocolo de asignación:
Un proceso solo puede pedir recursos (tipo) en orden creciente, si necesita uno de orden más bajo, debe liberar todos los recursos de orden más alto. Es decir:
Un proceso puede pedir un r(j) si y solo si $F(r(j)) > F(r(i))$ para todo i de los recursos que ya tiene. Si no cumple esta condición, debe liberar todos los r(i) que cumplen $F(r(i)) \geq F(r(j))$.
Dadas estas condiciones se puede demostrar que no tendremos nunca una espera circular.

DEMOSTRACION
Supongamos que ésta existe en el conjunto (p(0),p(1), ...,p(n)) o sea que p(i) está esperando r(i), quien está detenido por p(i+1) y por supuesto p(n) está esperando por un recurso retenido por p(0) (módulo n+1).
Como p(i+1) está reteniendo r(i) y esperando r(i+1) entonces $F(r(i)) < F(r(i+1))$, esto implica :
$F(r(0)) < F(r(1)) < ...... < F(r(n)) < F(r(0))$ ==> por transitividad $F(r(0)) < F(r(0))$ que es absurdo, luego es absurda la existencia de una espera circular.

17.6.2 - **Formas de EVITAR EL DEADLOCK**

Para evitar DEADLOCKS, en los casos en que se deba coexistir con las 4 condiciones necesarias, de debe conocer de qué manera los procesos requerirán sus recursos.
El modelo más sencillo es conocer de antemano la cantidad máxima de cada tipo de recurso a necesitar. Conocido esto es posible construir un algoritmo que permita que nunca se entre en "estado de deadlock". Consiste en examinar periódicamente el número de recursos disponibles y asignados para evitar una "espera circular".
Un estado "seguro" (**SAFE**) es cuando se pueden asignar recursos a cada proceso en algún orden y evitar el deadlock.
Formalmente: Un sistema está en estado "**seguro**" si existe una "**secuencia segura de procesos**". Una secuencia <p(1),p(2), ...,p(n)> es segura si por cada p(i) los recursos que necesitará pueden ser satisfechos por los recursos disponibles más todos los recursos retenidos por todos los p(j) tal que $j < i$. En este caso p(i) puede esperar que todos los p(j) terminen, obtener todos sus recursos y luego que los libera, el p(i+1) a su vez, puede obtener todos los recursos necesarios. Si esta secuencia no existe el sistema está en estado inseguro "**UNSAFE**".
Veamos un ejemplo:
Los procesos necesitarían cintas, existiendo un máximo de 12 cintas disponibles.
P(0) ----> 10   P(1) ----> 4   P(2) ----> 9
En un instante T(0) dado tienen asignado:
T(0)   P(0) <---- 5   P(1) <---- 2   P(2) <---- 2   3 Libres
Existe una secuencia segura <P(1),P(0),P(2)>.

	Necesita	Tiene
P(1)	4	2
P(0)	10	5
P(2)	9	2

P(1) toma 2 más. Cuando termina libera 4, más la instancia que quedaba libre son 5, luego
P(0) toma 5. Cuando termina libera las 10 retenidas y luego
P(2) toma 7 y luego termina.
Luego estamos en un estado seguro, con lo cual no podemos pasar a un estado de "deadlock" si se asignan los recursos en el orden dado por la secuencia dada.
De un estado "seguro" podemos pasar a uno "inseguro" (UNSAFE). Supongamos que en el instante T(1), se da un recurso más a P(2) :
T1)   P(0) <---- 5   P(1) <---- 2   P(2) <---- 3   2 Libres
Solo P(1) puede satisfacer sus requerimientos, aquí no existe una secuencia segura, pues ni P(0), ni P(2) pueden obtener la totalidad de recursos que requieren, entonces podemos desembocar en un estado de "deadlock". Si se hubiera respetado la secuencia "segura", esto no se habría producido.
Luego se pueden definir algoritmos que aseguren no entrar en un estado "inseguro", evitando en conclusión el estado de "deadlock".
Esto está indicando que cuando un recurso es pedido, aún estando disponible, es necesario verificar si no desembocará en un estado "inseguro".
CONCLUSIONES: * *Un estado de "deadlock" es un estado "inseguro".*

** Un estado "inseguro" puede desembocar en un "deadlock", pero no necesariamente lo es.*
** Si aún pidiendo un recurso, y estando este libre, se debe esperar, esto reduce la utilización del recurso.*

17.6.2.1 - **ALGORITMOS PARA EVITAR EL DEADLOCK** (Varias instancias por recurso)

El **Algoritmo del Banquero.** (The Banker's Algorithm - Dijkstra (1965) Habermann (1969)) se basa en determinar si los recursos que están libres pueden ser adjudicados a procesos sin abandonar el estado "seguro".

Supondremos N procesos y M clases de recursos y ciertas estructuras de datos

- **Disponible** : Vector de longitud M. Disponible(j) = k, indica que existen k instancias disponibles del recurso r(j).
- **MAX** : Matriz de NxM. Define la máxima demanda de cada proceso. MAX(i,j) = k, dice que p(i) requiere a lo sumo k instancias del recurso r(j).
- **Asignación** : Matriz de NxM. Define el número de recursos de cada tipo actualmente tomados por cada proceso. Asignación(i,j,) = k, dice que el proceso p(i) tiene k instancias del recurso r(j).
- **Necesidad** : Matriz de NxM. Define el resto de necesidad de recursos de cada proceso. Necesidad(i,j) = k significa que el proceso p(i) necesita k más del recurso r(j).
- **Requerimiento** : Es el vector de requerimientos de p(i). Requerimiento (i)(j) = k, indica que p(i) requiere k instancias del recurso r(j).

De lo cual se desprende que:

Necesidad(i,j) = MAX(i,j) - Asignación(i,j)

Para simplificar utilizaremos la notación siguiente:

Si X e Y son dos vectores:

$X \leq Y$ sii $X(i) \leq Y(i)$ para todo i y

$X < Y$ sii $X \leq Y$, y $X \neq Y$.

Además trataremos aparte las matrices por sus filas, por ejemplo, Asignación(i) define todos los recursos asignados por el proceso p(i).

Requerimiento(i) es el vector de requerimientos de p(i).

Requerimiento(i)(j) = k indica que p(i) requiere k instancias del recurso r(j).

ALGORITMO

Cuando se requiere un recurso se toman estas acciones:

1. Si Requerimiento(i) ≤ Necesidad(i) seguir, sino ERROR (se pide más que lo declarado en MAX(i)).
2. Si Requerimiento(i) ≤ Disponible seguir, si no debe esperar, pues el recurso no está disponible
3. Luego el sistema pretende adjudicar los recursos a p(i), modificando los estados de la siguiente forma:

Disponible = Disponible - Requerimiento(i)
Asignación(i) = Asignación(i) + Requerimiento(i)
Necesidad(i) = Necesidad(i) - Requerimiento(i)

Si esta nueva situación mantiene al sistema en estado "seguro", los recursos son adjudicados. Si el nuevo estado es "inseguro", p(i) debe esperar y, además, se restaura el anterior estado de asignación total de recursos.

17.6.2.2. - **ALGORITMO DE SEGURIDAD**

El algoritmo de seguridad nos permite determinar si un sistema está o no en estado seguro.

P1. Definimos **Work = Disponible** (de M elementos) y Finish = F (de Falso) para todo i con i=1,2,.....,n (Procesos).
P2. Buscamos la punta de la secuencia de 'SAFE' y los subsiguientes. Encontrar el i tal que:
a) Finish(i) = F y
b) Necesidad(i) ≤ Work
si no existe ese i , **ir a Paso P4**.
P3. Work = Work + Asignación(i)
Finish(i) = V (por Verdadero);
ir a Paso P2.
P4. Si Finish(i) = V para todo i, el sistema está en estado "SAFE", o sea, encontramos la secuencia de procesos.

Finish(i) va marcando cuáles son los procesos ya controlados (si están en V) y con Work vamos acumulando a los recursos libres que quedan a medida que terminan. Requiere MxNxN operaciones.

Ejemplo

Sea el conjunto de procesos {p(0),p(1),p(2),p(3),p(4)} y 3 recursos {A,B,C}. A tiene 10 instancias, B tiene 5 instancias y C tiene 7 instancias.

En el tiempo T(0) se tiene lo siguiente:

	Asignación	MAX	Disponible	Necesidad = MAX - Asig
	A B C	A B C	A B C	A B C
p(0)	0 1 0	7 5 3	3 3 2	7 4 3
p(1)	2 0 0	3 2 2		1 2 2
p(2)	3 0 2	9 0 2		6 0 0
p(3)	2 1 1	2 2 2		0 1 1
p(4)	0 0 2	4 3 3		4 3 1

El sistema está en estado seguro ya que la secuencia {p(1) ,p(3), p(4), p(2), p(0)} satisface el criterio de seguridad.

Supongamos ahora que p(1) requiere una instancia de A y 2 instancias de C, entonces: Requerimiento = (1,0,2).

Para decidir que puedo garantizar la satisfacción de este requerimiento:

a) Veo que Req(i) ≤ Disponible ((1,0,2) ≤ (3,2,2))

b) Veo que Req(i) ≤Necesidad ((1 0 2) ≤ (1 2 2))

c) Cumplimos el requerimiento, con lo cual se altera la información reflejando el siguiente estado:

	ASIG	NEC	NUEVO DISP.
p(0)	0 1 0	7 4 3	2 3 0
p(1)	3 0 2	0 2 0	
p(2)	3 0 2	6 0 0	
p(3)	2 1 1	0 1 1	
p(4)	0 0 2	4 3 1	

Y debemos verificar si es un estado seguro, para lo cual ejecutamos nuestro algoritmo de seguridad y hallamos la secuencia {p(1),p(3),p(4),p(0),p(2)}; por lo tanto puedo garantizar el cumplimiento del requerimiento de p(1).

Se puede ver que en este estado un requerimiento de p(4) de (3,3,0) no puede darse pues los recursos no están disponibles. Un requerimiento de p(0) de (0,2,0) no puede darse no porque no haya recursos disponibles, sino porque el estado resultante es inseguro.

Se prueba matemáticamente que si existe una secuencia segura entonces existen infinitas secuencias seguras.

17.6.2.3. - **Algoritmo para recursos de una sola instancia**

Ya que el algoritmo del banquero necesita m x n^2 operaciones, para recursos de una sola instancia usaremos un algoritmo más eficiente.

Además de los arcos de requerimiento (p(i),r(j)) y de asignación (r,(j),p(i)) agregamos uno de Pedido <p(i),r(j)> que son todos los recursos que en algún momento el proceso p(i) puede llegar a solicitar, y se demarcará por una flecha de tonalidad más débil.

Cuando un Pedido es requerido se transforma en Requerimiento.

Cuando un Asignado es liberado se transforma en Pedido.

El algoritmo dice: cuando un proceso p(i) requiere un recurso r(j), solo le será asignado si no se produce un ciclo en el grafo de adjudicación, pues si no hay ciclo, estamos en estado "SAFE". Si hay ciclo, estamos en estado "UNSAFE", luego hay posibilidad de Deadlock. Por ejemplo si p(2) pide r(2) :

Hay ciclo, estamos "UNSAFE", si p(1) pidiese r(2) y p(2) pidiese r(1), estaríamos en "Deadlock".

Requiere sólo del orden de NxN operaciones, donde N es el número de procesos (pues es el orden de operaciones de encontrar un ciclo en un grafo : N con N).

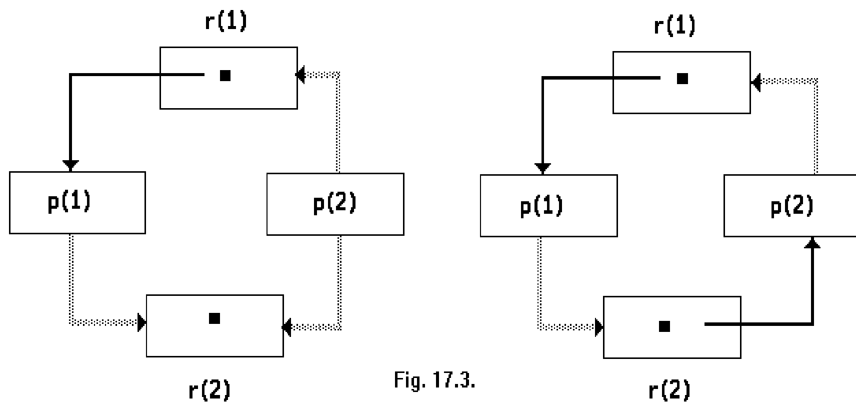


Fig. 17.3.

17.6.3 - **DETECCION DE DEADLOCK.**

Si no se tienen métodos que aseguren que el deadlock no ocurrirá, será necesario un esquema de detección y recuperación del 'deadlock'.

Se deberá :

a) Mantener información sobre la adjudicación de recursos y pedidos.

b) Tener un algoritmo que utilice esa información y determine si estamos en estado de 'Deadlock'.

Obviamente el algoritmo y el mantenimiento de la información incrementan el overhead que incluye no solo el tiempo de CPU, sino también el costo de mantener la información necesaria para la detección, la ejecución del algoritmo de detección y las pérdidas potenciales en caso de recupero ante deadlock.

EJEMPLO:

5 procesos P(0)....P(4), 3 recursos A, B, C. A tiene 7 instancias, B tiene 2 y C tiene 6.

	Asignación			Requerimiento			Disponible		
	A	B	C	A	B	C	A	B	C
P(0)	0	1	0	0	0	0	0	0	1
P(1)	2	0	0	2	0	2			
P(2)	3	0	2	0	0	0			
P(3)	2	1	1	1	0	0			
P(4)	0	0	2	0	0	2			

Decimos que el sistema no está en estado de deadlock. Si ejecutamos nuestro algoritmo encontramos la secuencia segura p(0), p(2), p(3), p(1), p(4) que nos da Finish(i) = Verdadero para todo i.

Si ahora modificamos que requerimiento de p(2) para el recurso pase a 4 (es decir que p(2) pida dos instancias más de C) el sistema está en deadlock. A pesar de poder utilizar los recursos asignados a p(0) esto no alcanza para satisfacer los requerimientos de los otros procesos. Luego existe un deadlock consistente en los procesos p(1), p(2), p(3) y p(4).

17.6.3.1 - **Recursos con varias instancias**

- **Disponible** : Vector de longitud M que indica los recursos disponibles de cada tipo.
- **Asignación** : Matriz de NxM (N procesos). Define los recursos tomados por cada proceso.
- **Requerimiento o Espera**: Matriz de NxM. Define los recursos requeridos por cada proceso en un instante, es decir, aquellos recursos por los cuales el proceso se encuentra en espera de disponibilidad.

La matriz de Requerimiento (o Espera) está indicando aquellas instancias por las cuales un determinado proceso se encuentra en espera.

Nótese que tiene sentido analizar para el caso de detección de deadlock aquellas instancias que se encuentran efectivamente asignadas a cada proceso (matriz Asig) y aquellas instancias por las cuales los procesos se encuentran actualmente en espera.

ALGORITMO DE SHOSHANI Y COFFMAN (1970)

P1. Work = Disponible

Sí Asignación distinto de 0 ==> Finish(i) = Falso sino Finish(i) = Verdadero

P2. Encontrar un i tal que

a) Finish(i) = F

b) Requerimiento(i) $\leq$ Work sino ir a **Paso P4**.

P3. Work = Work + Asignación

Finish(i) = V ir a **Paso P2**.

P4. Si Finish(i) = F entonces p(i) está en DEADLOCK.

17.6.3.2 - **Algoritmos para recursos de una sola instancia**

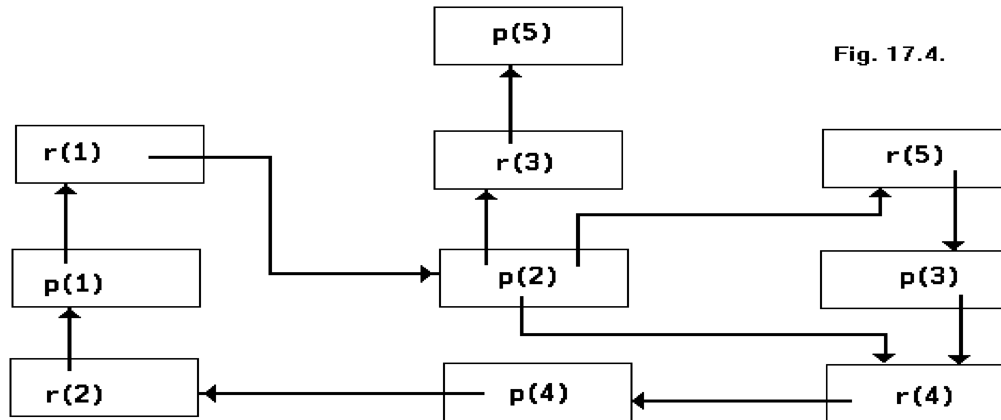
El anterior, como antes requiere MxNxN operaciones. En este caso veamos una variante del grafo de asignación de recursos, llamado GRAFO DE ESPERA, que se obtiene quitando del primero todos los recursos.

Un arco de p(i) a p(j) en un grafo de espera indica que el proceso p(i) está esperando que el proceso p(j) libere algún recurso que el primero necesita.

Un arco p(i),p(j) existe en un grafo de espera si y solo si en el correspondiente grafo de asignación existen dos arcos p(i), r(q) y r(q),p(j) para algún recurso r(q).

Veamos un ejemplo:

En el grafo de espera hay que detectar ciclos, lo que lleva NxN operaciones, donde N es el número de pro-



cesos.

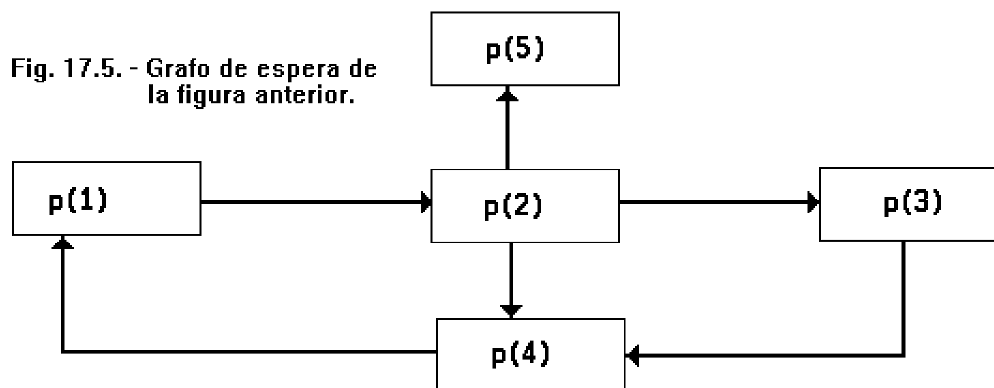


Fig. 17.5. - Grafo de espera de la figura anterior.

17.6.3.3 - **CUÁNDO SE APLICAN LOS ALGORITMOS ?**

Para saber cuándo es necesario aplicar el algoritmo de detección de "deadlock", es necesario saber:
a) La frecuencia de los deadlocks.
b) Cuántos procesos son afectados.
El caso extremo es llamarlo toda vez que un requerimiento no pueda inmediatamente ser satisfecho.
Un caso más simple es invocarlo cada x tiempo, por ejemplo 1 hora, o cuando la utilización del Procesador desciende su actividad por debajo de un determinado porcentaje de utilización, pues un 'deadlock' eventualmente disminuye el uso de procesos en actividad.
Si el algoritmo es invocado en forma arbitraria habrá muchos ciclos en el grafo de recursos, No podremos determinar cuál de los procesos que forman el ciclo es el que “causa” el deadlock.

17.7. - **RECUPERACION FRENTE DEADLOCK**

Frente al problema de deadlock, es necesario romperlo, de acuerdo a:
a) Violar la exclusión mutua y asignar el recurso a varios procesos
b) Cancelar los procesos suficientes para romper la espera circular
c) Desalojar recursos de los procesos en deadlock.

Para eliminar el deadlock matando procesos pueden usarse dos métodos:
- Matar todos los procesos en estado de deadlock. Elimina el deadlock pero a un muy alto costo.
- Matar de a un proceso por vez hasta eliminar el ciclo. Este método requiere considerable overhead ya que por cada proceso que vamos eliminando se debe reejecutar el algoritmo de detección para verificar si el deadlock efectivamente desapareció o no.

Esta eliminación puede no ser tan sencilla (un proceso que está a la mitad de la actualización de un archivo, un proceso que ya imprimió gran parte de un listado, etc.).
La cuestión es básicamente una cuestión de costo. Trataremos de eliminar a aquellos procesos que impliquen un “costo mínimo”. Este costo mínimo lamentablemente depende de varios factores:
- Prioridad de los procesos
- Cuánto hace que el proceso está trabajando y cuánto le falta para terminar.

- Cuántos y qué tipo de recursos usó el proceso.
- Cuántos recursos más necesita el proceso para terminar
- Cuantos procedimientos son necesarios para retrotraer al proceso a cero o a un estado anterior.

Algunas actividades necesarias para realizar estas cancelaciones serían:

1) **Selección de Víctimas**

Habría que seleccionar aquellos que rompen el ciclo y tienen mínimo costo. Para ello debemos tener en cuenta:
- Prioridad.
- Tiempo de proceso efectuado, faltante.
- Recursos a ser liberados (cantidad y calidad).
- Cuántos procesos quedan involucrados.
- Si por el tipo de dispositivo es posible volver atrás.

2) **Rollback**

- Volver todo el proceso hacia atrás (cancelarlo).
- Volver hacia atrás hasta un punto en el cual se haya guardado toda la información necesaria (CHECKPOINT)

3) **Inanición (starvation)**

Si el sistema trabajase con prioridades, podría ser que las víctimas fuesen siempre las mismas, luego habría que llevar una cuenta de las veces que se le hizo **Rollback** y usarlo como factor de decisión.

17.8. - **Conclusiones**

Determinar que se va a usar para evitar el deadlock depende de la clase de recursos (Howard 1973).
Es fácil ver que un sistema que emplee esta estrategia de “clases de recursos” no estará sujeto a deadlocks. Incluso produciéndose un deadlock, no involucrará más de una “clase”, ya que se utiliza la técnica de ordenamiento de recursos.
Dentro de cada clase se utiliza alguna de las técnicas descriptas. Por ejemplo consideremos un sistema con 4 clases de recursos:
- Recursos internos: utilizados por el sistema, por ejemplo BCP
- Memoria Central: memoria usada por el trabajo del usuario
- Recursos del trabajo: dispositivos y archivos
- Espacio de swapping: espacio del trabajo del usuario en almacenamiento secundario

Una solución ideal para el manejo de los deadlocks ordena las clases como sigue:

Bloque de control de Procesos - Numeración de recursos (se demostró que nunca se entra en 'DEADLOCK').
Memoria Central - Desalojo (Memoria Virtual, Swapping)
Cintas - Se puede evitar, ya que la información se puede obtener del lenguaje de comunicación del sistema operativo y asignado en forma total.
Espacio de Swapping (en disco) - Preasignación del espacio total necesario.

CAPITULO 18

PROTECCION Y SEGURIDAD

18.1. PROTECCION

Hasta ahora hemos visto elementos de protección que iban surgiendo según las necesidades de los mecanismos o administraciones que hemos ido estudiando. Estos elementos fueron: instrucciones privilegiadas, protecciones de memoria, operaciones de E/S realizadas por el Sistema Operativo, permisos de acceso a archivos, etc. Ahora trataremos de formalizar este concepto.

Los diversos procesos en un SO deben estar protegidos de las actividades de los otros. Con ese propósito se crearon diversos mecanismos, que pueden usarse para asegurar que los archivos, segmentos de memoria, CPU y otros recursos pueden ser usados solo por los procesos que tienen autorización del SO.

Por ejemplo, la facilidad de control de acceso en un sistema de archivos permite a los usuarios dictar cómo y quién puede acceder a sus archivos. El hardware de direccionamiento de memoria asegura que cada proceso solo ejecutará dentro de su espacio. El timer asegura que ningún proceso obtenga la CPU y no la libere. Finalmente, los usuarios no tienen permiso de hacer su propia E/S para proteger la integridad de los periféricos.

Examinemos en más detalle este problema, y construyamos un modelo único para implementar protección.

18.1.1. Objetivos de la protección

A medida que los sistemas se hacen más sofisticados, se hace necesario proteger su integridad. La protección, originalmente era un agregado a los sistemas operativos con multiprogramación, para que los usuarios pudieran compartir de forma segura espacio en común, como ser un directorio, o memoria. Los conceptos modernos de protección están relacionados con la posibilidad de aumentar la rentabilidad de los sistemas complejos que comparten recursos.

La Protección se refiere a un mecanismo para controlar el acceso de programas, procesos o usuarios a los recursos definidos por un sistema. Este mecanismo debe proveer los medios para especificar los controles a ser impuestos, junto con medios de refuerzo. Distinguimos entre protección y seguridad, que es una medida de la confianza de que la integridad de un sistema y sus datos serán preservados.

Hay varios motivos para la protección. El más obvio es la necesidad de prevenir la violación de una restricción de acceso por un usuario del sistema. Sin embargo es de mayor importancia la necesidad de asegurar que cada componente de programa activo en un sistema usa recursos de forma consistente con los permisos establecidos para el uso de esos recursos.

La protección puede mejorar la rentabilidad detectando errores latentes en interfases entre componentes de subsistemas. La detección temprana de ellos pueden prevenir la contaminación de un subsistema saludable por un sistema que no funciona bien. Un recurso no protegido no puede defenderse del uso de un usuario incompetente o desautorizado. Un sistema orientado a protección provee medios para distinguir entre el uso autorizado y el que no.

18.1.2. MECANISMOS Y POLITICAS

Un sistema puede verse como un conjunto de procesos y recursos. Para asegurar la operación eficiente y ordenada del sistema, los procesos están sujetos a políticas que gobiernan el uso de esos recursos. El rol de la protección es proveer un mecanismo para reforzar las políticas que gobiernan el uso de recursos. Estas políticas se pueden establecer en diversas formas. Algunas se fijan en el diseño del sistema, mientras que otras se formulan por la construcción de un sistema. Además hay otras definidas por los usuarios individuales para proteger sus propios archivos y programas. Un sistema de protección tiene que tener flexibilidad para permitir una variedad de políticas.

Las políticas para uso de recursos pueden variar, dependiendo de la aplicación, y están sujetas a cambio a través del tiempo. Por estas causas, la protección no se considera solo un problema del diseñador del sistema operativo. Tiene que ser una herramienta para el programador, de tal forma que los recursos creados y soportados por subsistemas de aplicación puedan cuidarse del mal uso. En este capítulo describimos los mecanismos que tiene que proveer el sistema operativo, de tal forma que los diseñadores de aplicaciones los puedan usar para diseñar su software de protección propio.

Es fundamental separar la Política del Mecanismo. El mecanismo determina como hacer algo, las políticas determinan que es lo que se hará, pero no como. Esta separación es muy importante por cuestiones de flexibilidad. Las políticas pueden cambiar de tiempo en tiempo, y en el peor de los casos, un cambio en la política requerirá un cambio en el mecanismo. Los mecanismos generales podrían ser más deseables, porque un cambio en la política solo requerirá la modificación de una serie de tablas o parámetros.

18.1.3. Dominios de protección

Un sistema de computación es una colección de procesos y objetos (de hardware - CPU, impresoras, etc. - y software - archivos, programas, semáforos, etc -). Cada objeto tiene un nombre único que lo diferencia de los demás objetos del sistema, y puede ser accedido solo a través de operaciones bien definidas y significativas. Los objetos son, esencialmente, Tipos Abstractos de Datos.

Las operaciones que son posibles pueden depender del objeto. Por ejemplo, una CPU solo puede ejecutar. Los segmentos de memoria pueden ser leídos o escritos, una lectora de tarjetas sólo puede ser leída. Lo mismo respecto de cintas, discos, archivos y otros dispositivos.

Obviamente, un proceso debería tener permiso de acceder solo a los recursos a los que tiene permiso. Más aún, en cualquier momento solo debe poder acceder a los recursos que necesita para su tarea. Por ejemplo, cuando el proceso p invoca al procedimiento A, el procedimiento solo debe poder acceder a sus variables y los parámetros formales que se le pasaron; no debe poder acceder a todas las variables de p. Similarmente consideremos cuando p invoca un compilador para compilar un archivo en particular. El compilador no debe poder acceder a cualquier archivo arbitrario, sino solo a un subconjunto bien definido de archivos (fuentes, listados, objetos, etc.) relacionados con el archivo a ser compilado. Inversamente, el compilador puede tener archivos privados usados para propósitos de optimización o estadísticas, que el proceso no debe acceder.

Para facilitar este esquema, introducimos el concepto de *Dominio de Protección*. Un proceso opera con un dominio de protección, que especifica los recursos a los que puede acceder el proceso. Cada dominio define un conjunto de objetos y los tipos de operaciones que se pueden invocar en cada objeto. La posibilidad de ejecutar una operación en un objeto es un derecho de acceso. Un dominio es una colección de derechos de acceso, cada uno de los cuales es un par ordenado { nombre de objeto, conjunto de derechos }. Si, por ejemplo, el dominio D tiene el derecho de acceso { archivo F, {lectura, escritura} }, entonces un proceso que ejecuta en el dominio D puede leer y escribir el archivo F, y no puede hacer ninguna otra operación en ese objeto.

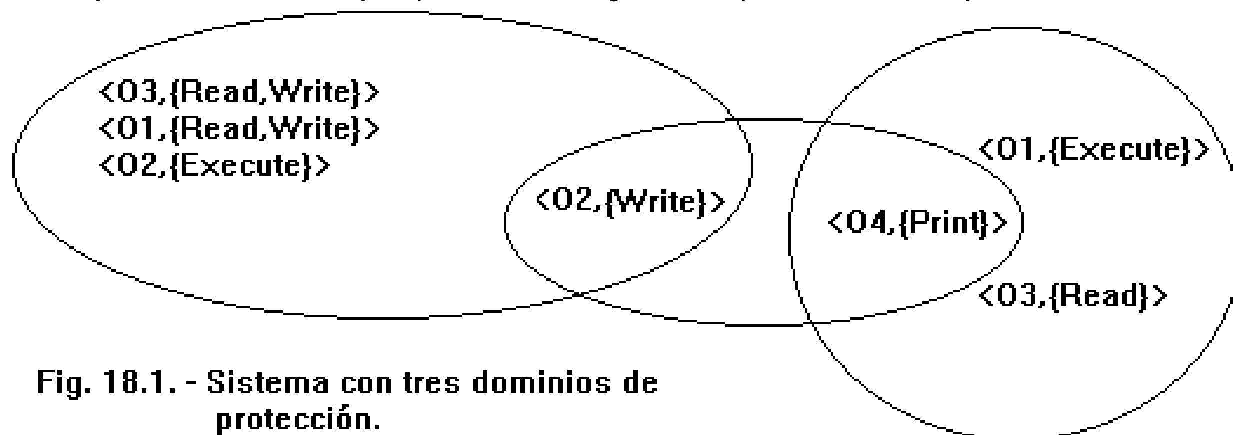


Fig. 18.1. - Sistema con tres dominios de protección.

Los dominios no necesitan ser disjuntos: pueden compartir los derechos de acceso. Por ejemplo, en la figura 18.1. tenemos 3 dominios: D1, D2 y D3. El derecho de acceso { O4, {print} } está compartido por D2 y D3, significando que un proceso ejecutando en cualquiera de estos dos dominios puede imprimir el objeto O4. Notar que un proceso debe estar ejecutando en el dominio D1 para leer y escribir en el objeto O1. Por otro lado, solo los procesos en el dominio D3 pueden ejecutar el objeto O1.

Consideremos el modo standard usuario/supervisor. Cuando un proceso ejecuta en modo supervisor, puede ejecutar instrucciones privilegiadas, y tomar control de todo el sistema. Por otro lado, si el proceso ejecuta en modo usuario, solo puede invocar instrucciones no privilegiadas. Estos dos modos protegen al sistema operativo (que ejecuta en modo supervisor) de los procesos de usuarios (que ejecutan en modo usuario). En un sistema operativo multiprogramado, dos dominios de protección son insuficientes, porque los usuarios quieren estar protegidos uno del otro. Es necesario un esquema más elaborado.

18.1.4. LA MATRIZ DE ACCESOS

Nuestro modelo de protección puede verse de forma abstracta como una matriz. Las filas de la misma representa los dominios, y las columnas, objetos. Cada entrada en la matriz consiste de un conjunto de derechos de acceso. Como los objetos están definidos explícitamente por la columna, podemos omitir el nombre del objeto del derecho de acceso. La entrada acceso(i,j) define el conjunto de operaciones que un proceso, ejecutando en el dominio Di, puede invocar sobre el objeto Oj.

Para ilustrar estos conceptos, la matriz de 18.2. tiene 4 dominios y 5 objetos: 3 archivos, una lectora y una impresora. Cuando un proceso ejecuta en el dominio D1, puede leer F1 y F3.

Objeto / Dominio	F1	F2	F3	Lector tarjetas	Impresora
D1	Read		Read		
D2				Read	Print
D3		Read	Execut		
D4	Read Write		Read Write		

Fig. 18.2. - Matriz de accesos.

18.1.5. Implementación de la Matriz de Accesos

Esta matriz generalmente es una matriz esparza, o sea que casi todas sus entradas están vacías. Existen varias implementaciones posibles:

18.1.5.1. TABLA GLOBAL

La implementación más simple de la matriz de accesos es una tabla global, consistiendo de un conjunto de ternas { dominio, objeto, conjunto de derechos }. Cuando se ejecuta una operación M en un objeto Oj dentro del dominio Di, se busca en la tabla { Di, Oj, Rk }, donde M pertenece a Rk. Si se encuentra la terna, se permite la operación, sino ocurre una excepción. Esta implementación tiene varias desventajas. La tabla suele ser muy grande, y no puede mantenerse en memoria por lo tanto existen más E/S. Además es difícil agrupar objetos o dominios con características similares. Por ejemplo., si un objeto puede ser leído por cualquiera, tiene que tener una entrada por separado en cada dominio.

18.1.5.2. LISTAS DE ACCESO

Cada columna en la matriz de accesos se puede implementar como una lista de accesos para un objeto. Las entradas vacías se obvian. La lista resultante para cada objeto consta de pares ordenados { dominio, conjunto de derechos }, que define todos los dominios con un conjunto no vacío de derechos de acceso para ese objeto.

Esta aproximación puede extenderse para definir una lista más un conjunto Default de derechos de acceso. Cuando la operación M en un objeto Oj es intentada en el dominio Di, buscamos la lista de accesos para el objeto Oj, y miramos una entrada { Dj, Rk } con M perteneciente a Rk. Si se encuentra, permitimos la operación; si no miramos el conjunto default. Si M esta, también permitimos el acceso, sino, se niega el acceso.

18.1.5.3. LISTAS DE CAPACIDADES

Aquí el orden está dado por las filas de la matriz, donde cada fila es para un dominio.

Una **lista de capacidades** para un dominio es una lista de objetos, y las operaciones permitidas sobre él.

Un objeto, en general, está representado por su nombre o dirección (llamado capacidad).

Para ejecutar una operación r sobre un objeto O(j), el proceso invoca r especificando la "capacidad" (pointer) para el objeto O(j) como un parámetro. La posesión de la capacidad hace posible su acceso.

Este sistema es muy seguro si no se permite el acceso directo de la "capacidad" en el espacio de dirección del usuario (o sea, no tiene acceso a ella, por lo tanto no la puede modificar en forma directa).

Si todas las "capacidades" están protegidas, el objeto al cual protegen está seguro.

Las capacidades se distinguen de otros datos (tipos) por una de estas dos razones:

- Cada objeto tiene un rótulo (tag) para denotar su tipo como capacidad o como dato accesible. Los rótulos no deben ser accesibles por los programas de aplicación. Se usa al Hardware o al Firmware para hacer esto más fuerte. El hardware debe distinguir los tipos de datos (enteros, punteros, instrucciones, puntos flotantes, etc.).
- El espacio de direcciones de un programa debe ser dividido en dos partes: la de código y datos (accesible por el mismo), y la parte de lista de capacidades, accesible solo por el Sistema Operativo.

Ejemplos:
Un ejemplo serían los semáforos, a los que solo es posible acceder por medido de operadores P y V.
Una forma de implementación sería la de memoria segmentada.
Otro ejemplo es una L.C.U. del Administrador de Información.

18.1.5.4. IMPLEMENTACION DEL MECANISMO DE LOCK/KEY

El mecanismo de Lock/key es un compromiso entre listas de acceso y listas de capacidad.
Cada objeto tiene una lista de bits de candado (locks), y cada dominio tiene otra lista de bits de llave (keys).
Un proceso solo puede ejecutar (acceder) al objeto si el dominio al cual pertenece tiene llaves que coincidan con los candados del objeto.
Las llaves del dominio son manejadas solo por el sistema operativo.
Un ejemplo de esto es el Storage Control Key (deben coincidir los bits de la PSW o del canal con los bits de la página cuando no se usa el DAT).

18.1.6. Comparación de las implementaciones

Las Listas de Acceso corresponden a las necesidades de usuario. Cuando se crea un objeto especifica a qué dominio pertenece. En sistemas grandes, la búsqueda puede llegar a ser muy grande.
Las Listas de Capacidades no son sencillas de implementar por el usuario. Es eficiente una vez implementado, pues solo es necesario verificar que la capacidad es válida. La revocación es muy complicada, porque las capacidades están distribuidas por todo el sistema (recordemos L.C.U. de Administración de Información).
El mecanismo de Llave/Candado es una solución de compromiso. Es efectivo y flexible. Si bien cualquier cambio solo requiere cambiar la configuración de unos pocos bits, si el sistema posee una gran volatilidad (muchos dominios u objetos nuevos o que desaparecen, muchos permisos que cambian o se agregan constantemente) puede ser necesario replantear la función de mapeo lo cual, en estos casos, torna el mecanismo muy complejo.

18.1.7. Estructuras de protección dinámicas

La asociación entre un proceso y un dominio puede ser estática si los recursos disponibles para ese proceso lo serán para toda su ejecución, o dinámica si existen cambios de accesos o dominios. Si se permiten estos cambios, posiblemente sean violadas las protecciones.
Un mecanismo que nos permite implementarlo es incluir a los mismos dominios como objetos de la matriz de accesos, y cuando sean necesarios cambios en los accesos incluimos a la propia matriz como objeto. Como lo que se quiere es que cada entrada pueda ser modificada en forma individual, consideraremos cada entrada como un objeto.

18.1.8. Cambio de Dominio

SWITCH.
Un proceso puede cambiar del dominio D(i) al dominio D(j) sii el derecho de acceso SWITCH pertenece a Acceso(i,j).
Ejemplo:

Dominio \ Objeto	F1	F2	F3	Lector tarjetas	Impresora	D1	D2	D3	D4
D1	Read		Read				Swit.		
D2				Read	Print			Swit.	Switch
D3		Read	Execut						
D4	Read Write		Read Write			Swit.			

Fig. 18.3. - Matriz de la figura 18.2 con Dominios como Objetos.

Luego: D(2) puede cambiar a D(4) o D(3)
D(4) puede cambiar a D(1) pero no a D(2)
D(1) puede cambiar a D(2) pero no a D(3).

18.1.7. Cambio de contenido de la matriz de accesos

El permitir cambios controlados en la matriz de acceso requiere de tres operaciones adicionales: copy, owner y control

COPY

La capacidad de copiar un derecho de acceso existente de un dominio a otro (fila) de la matriz de accesos se indica agregando un asterisco al derecho de acceso.

El derecho COPY solo permite copiar el derecho de acceso dentro de la misma columna (es decir para el mismo objeto) en la cual está definido tal derecho.

Dominio \ Objeto	F(1)	F(2)	F(3)
D1	Read		Write *
D2		Read *	Ejecutar
D3	Read		

Fig. 18.4.

El ejemplo anterior se transforma en :

Dominio \ Objeto	F(1)	F(2)	F(3)
D1	Read		Write *
D2		Read *	Ejecutar
D3	Read	Read	

Fig. 18.5.

(*) Habilita el copiado del "acceso" en la misma columna (objeto)

Un proceso ejecutando en D(2) puede "copiar" el acceso Read en cualquier entrada asociada a F(2).

Algunas variantes de esto podrían ser:

- Transfer: se pasa de acceso(i,j) al acceso(k,j) pero se quita el acceso(i,j) (Removed).
- Copia limitada: se pasa solo el acceso R y no la capacidad R*.

OWNER

Necesitamos además del copiado algún mecanismo para agregar nuevos derechos y eliminar otros, el derecho de acceso OWNER controla estas operaciones.

Si en acceso(i,j) tenemos Owner, significa que un proceso ejecutando en D(i) puede agregar o quitar accesos en toda la columna j.

Ejemplo:

Dominio \ Objeto	F(1)	F(2)	F(3)
D1	Read Owner		Write
D2		Read * Owner	Read * Owner
D3	Read		

Fig. 18.6.

Se transforma en :

Dominio \ Objeto	F(1)	F(2)	F(3)
D1	Read Owner		
D2		Read/Write Owner *	Read Owner *
D3		Write	Write

Fig. 18.7.

CONTROL

El Copy y el Owner permiten que un proceso altere entradas en una columna. Un mecanismo para alterar entradas en las filas es asimismo necesario. Tal mecanismo es el derecho CONTROL que habilita cambios (remover derechos) en otras filas (dominios)

Dominio \ Objeto	F(1)	D(1)	D(2)	D(3)
D(1)	Read		Switch	
D(2)	Read			Switch Control
D(3)	Read/Write	Switch		

Fig. 18.8.

El ejemplo anterior se transforma en :

Dominio \ Objeto	F(1)	D(1)	D(2)	D(3)
D(1)	Read		Switch	
D(2)	Read			Switch Control
D(3)	Read	Switch		

Fig. 18.9.

O sea que D(2) puede cambiar el modo de acceso a F(1) de D(3) de R/W a R.

Lo importante de esto es que con estos esquemas y los conceptos de objetos y capacidad, es posible crear un nuevo tipo de monitor, llamado **manager**, que es usado para cada recurso, el cual planifica y controla el acceso a ese recurso. Cuando un proceso necesita un recurso, llama al manager, el cual le devuelve la capacidad para ese recurso. El proceso debe presentar la capacidad cuando usa el recurso. Cuando el proceso finaliza el uso del recurso devuelve la capacidad al manager, quien lo asignará a otro proceso, de acuerdo a su planificador (scheduler).

18.1.8. Revocación

En Protección Dinámica, cuando se revocan los accesos podemos tener varias opciones:

- Inmediata / Postergada.
- Selectiva / General (para algunos usuarios, o para todos).

- Parcial / Total (todos los accesos a un objeto, o solo algunos).
- Temporario / Permanente (se podrá obtener nuevamente o no).

En la lista de accesos la revocación es fácil. Se busca la lista de accesos y se hacen los cambios, luego cualquier combinación anterior es posible.

En el esquema de Listas de Capacidad, la revocación es más dificultosa, pues las capacidades están distribuidas por todo el sistema. Veamos los distintas sistemas de revocación en Listas de Capacidad:

- *Readquisición:* las capacidades son borradas periódicamente. Si un proceso intenta readquirirla, y la capacidad se revocó, NO puede hacerlo.
- *Back-pointers:* una lista de punteros asocia a cada objeto con todas las capacidades asociadas. Siguiendo los punteros es posible cambiarlos. La implementación es general pero muy costosa. Es utilizado en Multics.
- *Indirección:* las capacidades no apuntan al objeto en forma directa, sino a una única entrada en una tabla global, la cual, a su vez, apunta al objeto. La revocación se implementa buscando en la tabla global la entrada y borrándola. Puede ser reutilizada para otra capacidad. No se puede hacer revocación selectiva.
- *Llaves:* la revocación se hace cambiando la llave. Si un objeto tiene asociadas varias llaves es posible hacer revocación selectiva.

18.1.9. SISTEMAS EXISTENTES

UNIX

Cada archivo tiene asociado 3 campos (dueño, grupo, universo). Cada campo tiene 3 bits: R (read), W (write), X (execution).

Un dominio está asociado con el usuario.

Hacer switch de dominio corresponde a cambiar la identificación del usuario temporariamente. Cuando un usuario A comienza ejecutando un archivo de dueño B, el usuario es "seteado" a B, y cuando termina, es "reseteado" a A.

MULTICS

Su sistema de protección está sobre su sistema de archivos (todo es archivo), y cada archivo tiene asociada una lista de acceso, como el acceso del dueño y del universo.

Además tiene una estructura de anillos que representan los dominios, tal que dados los dominios D(i) y D(j), con i menor j, D(i) tiene mayores privilegios que D(j). Dados solo dos dominios, nos encontraríamos frente a un sistema maestro/esclavo. D(0) tiene el mayor privilegio de todos.

El espacio de direcciones es segmentado, y cada segmento tiene asociado su número de anillo y sus bits de acceso (R,W,X). Cuando un proceso ejecuta en D(i) no puede acceder a un segmento de D(j) si j menor i pero sí a una de D(k) si i menor = k, respetando, desde ya, los bits de acceso.

Para llamar procesos de anillos inferiores, se produce un trap (SVC), entonces es más controlado.

Protecciones de la forma "capacidad" los tenemos en sistemas como el Hydra y el Cambridge CAP System.

18.2. SEGURIDAD

La **seguridad de datos** estudia cómo proteger los datos de un sistema de computación contra accesos no autorizados, permitiendo, obviamente, los autorizados.

Existen diversos contextos en los cuales es necesaria la seguridad de datos:

- Sistemas de información y de archivos: controles de acceso (lo visto en Protección).
- Telecomunicaciones y redes de computadoras: controles criptográficos.
- Bases de Datos: controles sobre la información, búsquedas estadísticas y controles de inferencia.

En este apunte se tratan los dos últimos temas. La seguridad en Sistemas de Información ya fue tratada en el capítulo de Protección.

Un sistema de seguridad depende de que se pueda identificar correctamente al usuario, lo que se hace por medio de un protocolo de conexión (Login). El usuario se identifica, y el sistema le pide una palabra clave (password). Para ser almacenada, la palabra clave debe ser encriptada, porque sino sería necesario proteger el archivo que las contiene. La palabra clave que escribe el usuario se protege de su visualización eliminando el eco de lo escrito (shadow); luego se encripta y se la compara con la almacenada en el archivo. Si ambas coinciden, se permite el acceso; sino se le niega.

18.2.1. Principios de diseño para Sistemas de Seguridad

* Privilegio mínimo: sólo se deben otorgar los derechos necesarios para las funciones necesarias, y no más.

* Economía de mecanismos: los mecanismos de seguridad deben ser implementados en un solo módulo pequeño, que debe ser incluido en el diseño inicial del Sistema (sino se hiciera así, sería posible evitarlo).
* Mediación completa: debe controlar todos los accesos y debe ser imposible de obviar.
* Diseño abierto: la seguridad no debe depender de que los mecanismos sean secretos: solo deben serlo las claves.
* Aceptación psicológica: deben ser de uso sencillo, porque sino se corre el riesgo de que los usuarios no protejan sus datos.

18.2.2. Seguridad en Telecomunicaciones o Redes de Computadoras

"Si no hay comunicación, en principio hay privacidad; luego, el problema no existe". Cuáles son algunos de estos problemas ? :

- PRIVACIDAD: Es necesario mantener los datos en secreto de un punto al otro de la comunicación.
- AUTENTICIDAD: Es necesario conocer si los datos son auténticos o si fueron modificados antes de llegar.

Por ejemplo, en la Figura 18.10, M podría ser un espía industrial o financiero. También podría hacer cosas como generar o repetir mensajes de depósitos en cuentas corrientes (esto se podría evitar teniendo en cuenta el número de mensaje, la hora, la identificación de la terminal, etc.). Una solución para este tipo de problemas es la CRIPTOGRAFIA.

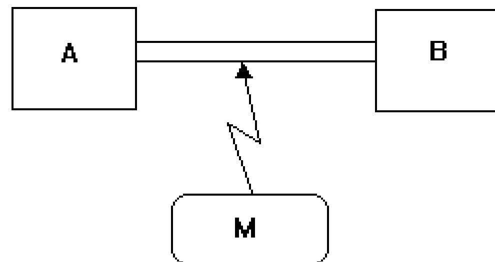


Fig. 18.10. - Escucha, genera o modifica datos.

La base de la criptografía es la siguiente:

Fig. 18.11.

(*) Se aplica un método y una clave (generalmente depende de que la clave sea desconocida).
Para realizar esto existen métodos clásicos de sustitución:

18.2.2.1. ENCIFRAMIENTO DE CESAR (SUSTITUCION)

Se suma un número entero fijo a las letras del alfabeto, por ejemplo + 3. Entonces, un texto llano, por ejemplo CESAR se convierte en FHVDU. Luego, el lugar de recepción debe restar 3 a lo recibido.

18.2.2.2. SUSTITUCION CON PALABRA CLAVE

Se escriben las letras del alfabeto mezcladas con una palabra clave, por ejemplo:
SIMON BOLIVAR. Sin letras repetidas: SIMON BLVAR.

Alfabeto	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Alf. Transformado	S I M O N B L V A R C D E F G H J K P Q T U W X Y Z

Aquí, CESAR = MNPSK.

Estos dos primeros tipos de encriptamiento que pueden ser resueltos fácilmente. Son de fácil deducción, porque hay mucha información que facilita la misma. En muchos idiomas se conoce la frecuencia aproximada de todas las letras en todas las palabras del idioma, y de esta forma el mensaje puede ser desencriptado fácilmente. Si se conoce parte del mensaje (por ejemplo: Logon), la tarea es más fácil aún.

Para mayor información acerca de como deducir claves de este tipo, y para regocijo del lector, léase "El escarabajo de oro", cuento de Edgar Allan Poe, que contiene suficiente información sobre el tema.

18.2.2.3. TRANSPOSICION (DES)

Consiste en permutar las letras de los mensajes, e inclusive hacerlo a nivel de bits.
Un método muy sencillo es usar el O exclusivo. La sutileza radica en la clave: se conoce el método pero no la clave:

XOR	0	1
0	0	1
1	1	0

Se aplica según el siguiente ejemplo:

TEXTO	CLAVE	CRIPTOGRAMA	CLAVE	TEXTO
1	1	0	1	1
1	0	1	0	1
0	1	1	1	0
1	0	1	0	1
0	1	1	1	0

En 1976 se estableció el DES (Data Encryption Standard) como norma para mensajes no relacionados con la seguridad nacional de los EEUU, por la NBS (National Bureau of Standards), y fue desarrollado por IBM. Consiste en dividir los datos en bloques de 64 bits, utilizando 56 para datos, y los otros 8 como paridad. Luego, existen 2^{56} claves posibles (2^{56} es aproximadamente igual a 7.2 E+16).

Supongamos tener una computadora capaz de analizar 10^6 claves por segundo:

1E+ 6 claves 1 seg.

7.2E+16 claves 7.2E+10 seg.

1 año = 31536000 seg = 31536 * 10^3 seg. Luego, tardaríamos

7.2E+10 / 31536 * 10^3 = **2283 años**

para obtener todas las claves posibles. Digamos que, en promedio, la hallaríamos en 1000 años.

Además, este es un mecanismo muy fácil de implementar por Hardware.

18.2.2.4. ONE-TIME PAD (BLOQUE DE USO UNICO)

Es un método totalmente seguro. Trabaja como el método de Cesar, pero el entero a sumar varía para cada caracter del texto en forma aleatoria.

Es seguro, porque el criptograma "SECRETO" podría provenir de:

MENTIRA con la clave 6 0 15 2 22 23 12, o de:

REALEZA con la clave 1 0 2 6 0 6 12.

El problema es que la clave es tan larga como el mensaje, y nunca debe ser reutilizada.

18.2.2.5. DISTRIBUCION DE CLAVES

Los métodos criptográficos (DES y One-time Pad) tienen el problema de la distribución de las claves. Todos los que participan deben conocerlas.

El problema reside en que las claves deben ser cambiadas con cierta frecuencia; en consecuencia se necesita un canal de distribución de claves. No debería usarse la misma red, por problemas de seguridad; hay que buscar otra equivalente. Esto puede parecer un problema insoluble. En general se termina utilizando un mensajero de confianza, pues se llega a la conclusión de que el cambio será esporádico.

18.2.2.6. CLAVES PUBLICAS

Los métodos tradicionales se denominan simétricos, pues usan la misma clave para encriptar y desencriptar. Los denominados asimétricos tienen dos claves: una para encriptar y otra distinta para desencriptar.

Supongamos dos interlocutores A y B. A usa una clave E para encriptar, y B usa una clave D para desencriptar, de tal manera que aunque se conozcan E y el método, no se pueda deducir D en un tiempo razonable (por ejemplo, que para deducir D se tarden miles de años).

Esto permite que A y B publiquen sus claves de encriptamiento Ea y Eb, y mantengan en secreto sus claves de desencriptamiento Da y Db. Con esto, el problema de distribución de claves desaparece.

Obviamente, el problema radica en generar claves E y D.

Uno de los métodos más importantes es el RSA (por sus autores Rivest, Shamir, Adleman).

La seguridad de este método depende de ciertas propiedades de los números primos, y de la dificultad de encontrar divisores de números muy grandes (cientos de dígitos).

RSA:

Se buscan dos primos grandes p y q, y se obtiene n = p.q .

Se buscan dos números e y d que contemplen la siguiente propiedad:

$e.d \bmod [(p-1).(q-1)] = 1$

Se representa el texto llano por medio de un M tal que

$0 \leq M \leq n-1$

de forma tal que el encriptamiento resulta:

$C = M^e \bmod(n)$

y el desencriptamiento es:

$M = C^d \bmod(n)$

Veamos un ejemplo con números pequeños:

p=3; q=5; n=15

Se elige e = 3 y d = 11 tal que 3.11 mod [2.4]=33 mod(8) = 1

Si suponemos un mensaje M = 8, resulta:

$c = 8^3 \bmod (15) = 512 \bmod(15) = 2$

siendo c = 2, el mensaje cifrado que viaja por el medio de comunicación. En el lugar de destino, se hace:

$M = 2^{11} \bmod(15) = 2048 \bmod(15) = 8$

donde M=8 es el mensaje original desencriptado.

La clave pública es el par (e,n) y la privada el par (d,n). Conocer e y n no da suficiente información como para hallar d (p y q son secretos).

Un espía lo que puede hacer es factorizar n, que es público, para obtener p y q, y luego podrá calcular d, ya que e es también público.

Pero el problema es que no se conocen algoritmos eficientes de factorización. Si p y q tienen 100 dígitos c/u, el mejor algoritmo de factoreo tardaría miles de millones de años (existen del orden de 10^{97} números primos de 100 dígitos).

El mejor algoritmo de factoreo, en la franja de 10^{200}, es el de Schroeppel (no publicado), del cual se han calculado algunos tiempos:

dígitos n = p * q	Tiempo (1 ms por operación aritmética)	
50	3.9	horas
75	104	días
100	74	años
200	3.8 E+9 años	
300	4.9 E+15	años
500	4.2 E+25	años

Justificando el RSA

Elegidos p y q primos, y n = p.q, se eligen e y d tal que

$e \cdot d \bmod [(p-1)(q-1)] = 1$

Dado un mensaje M / $0 \leq M \leq n$, debe ocurrir que:

$P(M) = M^{e} \bmod n = C$ y

$S(C) = C^{d} \bmod n = M$

o sea que esto funciona si S o P = Identidad, o sea,

$S(P(M))= M$ con $0 \leq M \leq n$.

Si escribimos

$M = S(P(M)) = (M^{e} \bmod n)^{d} \bmod n = M^{ed} \bmod n$, que es lo que habría que probar.

Para esto usamos el Teorema de Fermat, que dice:

Si p es primo, y M un entero no múltiplo de p, entonces para todo natural r,

$M^{r} \bmod p = M^{r \bmod(p-1)} \bmod p$

En nuestro caso es:

$e * d \bmod [(p-1)(q-1)]= 1$ (por construcción)

que se puede escribir:

$e * d = 1 + K (p-1).(q-1)$ para algún K

(esto sale de $e * d / [(p-1)(q-1)] = K$ con resto 1).

Si le aplicamos mod (p-1) en ambos miembros, resulta que:

$e * d \bmod (p-1) = 1 + 0$ [1]

pues K (p-1) (q-1) es múltiplo de (p-1).

Si M no es múltiplo de q, y por el teorema, tenemos:

$M^{ed} \bmod p = M^{ed \bmod(p-1)} \bmod p = M^{1} \bmod p$, por [1].

Si M es múltiplo de p, M^{ed} también lo es, por lo tanto:

$M^{ed} \bmod p = M \bmod p$ (pues obviamente es 0)

Entonces, $M^{ed} = M + s.p$ para algunos enteros s.

Análogamente

$M^{ed} = M + t.q$ para algunos enteros t; luego

$M^{ed} - M = s.p = t.q.$

Como p y q son primos distintos, y como p divide a s.p entonces p también divide a t.q; luego, divide a t entonces t = w.p. O sea:

$M^{ed} - M = w.p.q$ para algún w.

Luego $M^{ed} = M + w.n$ (p.q = n).

Entonces, $M^{ed} \bmod n = M \bmod n = M$ (pues M menor = n), o sea, S(P(M)) = M

En forma inmediata obtenemos:

$P(S(M)) = (M^{d} \bmod n)^{e} \bmod n = M^{ed} \bmod n = S(P(M)).$

O sea, que RSA es un sistema criptográfico conmutativo.

Si bien el RSA es un método laborioso, resuelve el problema de la distribución de claves, y permite la utilización de otros métodos. Por ejemplo:

A quiere conversar con B y genera una clave S de DES. A calcula S' = RSA(S), usando la clave pública de B, y envía S'. B desencripta S' con su clave privada y la transforma en S y responde a A con la clave S en encriptado DES.

18.2.2.7. AUTENTICIDAD (FIRMA)

Si pensamos el RSA al revés, por ejemplo, si encriptamos con una clave D secreta, se puede desencriptar con una clave E pública, y nadie podría falsificar el mensaje, ni siquiera el receptor. Esto podría usarse como "firma" de documentación electrónica.

18.2.2.8. REDES LOCALES

Estas redes tienen problemas de seguridad, pues son pensadas como "broadcast" (todos escuchan). Inclusive existe la posibilidad de daño físico, pues si alguien conecta el cable de transmisión al toma de corriente eléctrica, es posible dañar la interfase de nodos y hacerla inoperante.

18.2.3. Seguridad de datos en Bases de Datos

Si suponemos un acceso público a ciertas Bases de Datos, habría que mantener ciertos controles sobre su actualización. Por ejemplo:

Dependiente de los datos: solo es posible borrar registros que no hayan sido actualizados en por lo menos más de tres meses.

Dependiente del tiempo: el campo de sueldo solo es posible de ser cambiado en horarios determinados.

Dependiente del contexto: solo se pueden listar nombres o sueldos, pero no ambos a la vez.

Generalmente las bases de datos comerciales chequean con una Lista de Control de Acceso asociada a una relación o conjunto de campos o registros.

Además es necesario un control de consistencia de datos constante y antes de permitir un acceso o modificación.

En los casos que se permite el acceso a Bases de Datos con fines estadísticos no debería permitirse el acceso a datos individuales. O sea, no permitir preguntas del tipo: ¿Cuánto gana el Sr. X?. Pero si se conocen algunas características del sujeto, esto solo no basta para proteger la información individual. Supongamos que se sabe que el Sr. X es una Sra., y que es la única integrante femenina de un área determinada de una firma, es válido preguntar cual es el promedio de sueldo del personal femenino de esa área.

Por lo tanto habría que bloquear el acceso a la información cuando los datos recuperados son pocos; pero aún así se podría deducir algo si se pregunta: ¿Cuál es el sueldo promedio?, y luego, ¿Cuál es el sueldo promedio de los hombres?. Estos casos son muy difíciles de detectar.

18.2.4. Seguridad de Datos en general

Los casos más dañinos son los que se producen por descuido o malicia del personal de la misma instalación. Por ejemplo:

- Bomba de tiempo: cuando el programador es despedido, deja códigos que se destruyen o destruyen información a partir de una fecha.
- Redondeo de centavos: embolsar en una cuenta particular los centavos sobrantes por redondeo.
- Caballo de Troya: hacer copias falsas de pantallas de Login para capturar palabras claves.
- Basureros: buscar en los cestos de basura listados de directorios con palabras claves.
- Descuido: aprovechar que la terminal quedó conectada mientras el operador se fue, o utilizar palabras claves sencillas de adivinar.

CAPITULO 19.

PROCESOS Y PROGRAMACION CONCURRENTES.

19.1. SISTEMAS DE TIEMPO REAL

Como ya fue visto anteriormente un sistema de tiempo real generalmente se usa como un dispositivo de control en una aplicación dedicada.

La característica mas importante de estos sistemas es que tienen restricciones de tiempo bien definidas, y el procesamiento tiene que hacerse dentro de ese tiempo.

Existen dos clases de tales sistemas, a saber:

A) De reserva de pasajes, de Transacciones bancarias, (etc.) (Terminales ON-LINE).

B) De Control de procesos, que son sistemas, que estimulados por un evento externo, deben emitir una respuesta en un tiempo finito y determinado.

La diferencia fundamental entre ambas clases estriba en que en la primera el tiempo de respuesta no es crítico (con ciertas reservas) en tanto que en el segundo caso el tiempo de respuesta es sumamente crítico.
Asimismo puede decirse que es también un factor diferenciativo la administración de las interrupciones asociada con las prioridades. En la Fig. 19.1 se puede visualizar un esquema de un sistema de tiempo real.

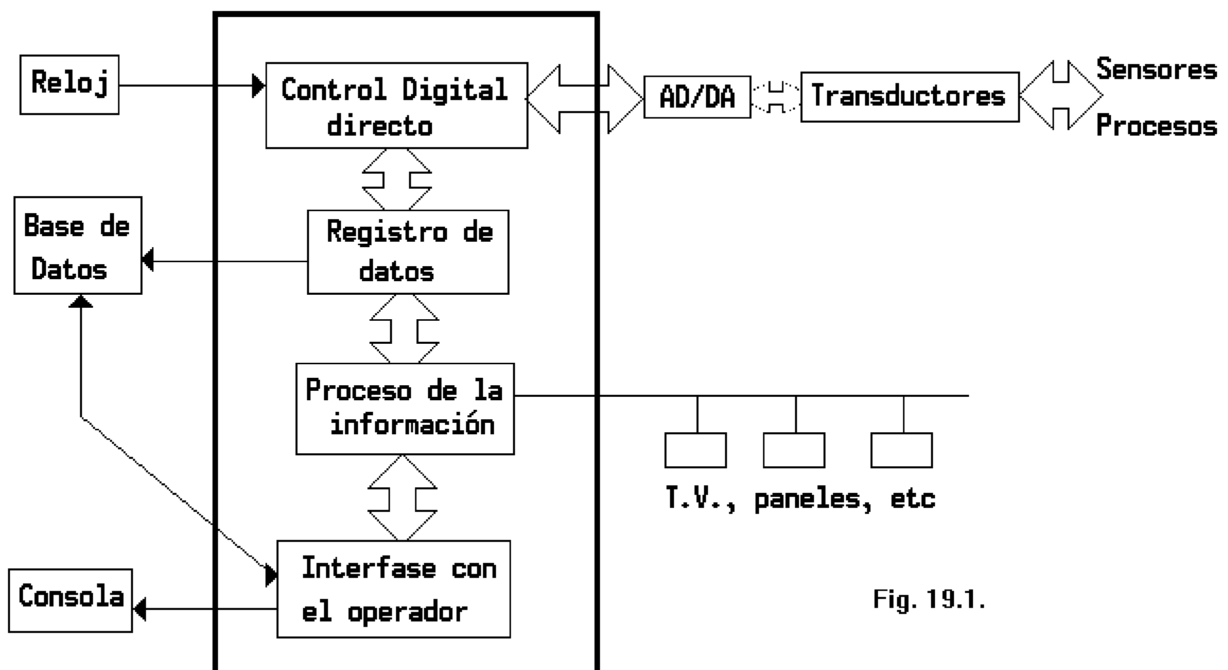


Fig. 19.1.

Las "funciones" del software serían tareas (ejecutándose en paralelo) que deben interactuar y sincronizarse para la transferencia de la información.

Una breve descripción del hardware incluiría:

- ADQUISICION DE DATOS:
  - Sensores: Elementos que alteran sus características en función de lo que se está midiendo.
  - Transductores: Realizan una traducción de un cambio físico a otro. (de algo a variaciones de corriente o tensión).
  - Conversores (AD/DA): (analógico digital y digital analógico) transforman variaciones de corriente o tensión en datos binarios.
  - Puertos: Serializan y deserializan la información.
- TAREAS DE CONTROL DIGITAL DIRECTO
  - Procesador: Suficientemente rápido para que las tareas sean ejecutadas en el tiempo adecuado.
- ACTUACION SOBRE EL PROCESO: Es igual que la parte de Adquisición de Datos pero a la inversa, cambiando Sensores por Actuadores.
- DETECCION DE EVENTOS: Mecanismo de interrupciones de varios niveles, con un esquema de prioridades asociado y posibilidad de enmascaramiento.
- ALMACENAMIENTO DE DATOS: Disco (datos al momento) y Cintas para históricos.
- INTERFASE CON EL OPERADOR: Impresoras, T.V., Hardcopy, señales, alarmas, etc.
- RELOJ: Marcador de secuencia de interrupciones.
- AUTODIAGNOSTICO

- HARDWARE QUE PERMITA: alocaciones dinámicas y procedimientos reentrantes.
Una descripción del software contendría:
- NUCLEO DEL SISTEMA OPERATIVO:
    - Administración del procesador o procesadores de manera de compartirse entre las tareas activas.
    - Soporte para tareas concurrentes bajo un esquema de prioridades
    - Soporte de comunicación y sincronización entre tareas para permitir que cooperen entre sí

19.2. - **INTRODUCCION A LA PROGRAMACION CONCURRENTE**

De una manera general los programas pueden ser clasificados en secuenciales y en concurrentes.

Un programa secuencial se caracteriza por no depender de la velocidad de ejecución y de producir el mismo resultado para un mismo conjunto de datos de entrada.

En un programa concurrente (o paralelo) las actividades que lo constituyen están relativamente superpuestas en el tiempo. Esto significa que una operación puede ser iniciada en función de la ocurrencia de algún evento, antes del término de la operación que estaba ejecutándose anteriormente.

Los elementos que constituyen un programa concurrente son módulos independientes denominados tareas o procesos.

La programación concurrente se encarga fundamentalmente de temas vinculados a la comunicación y sincronización de procesos.

La comunicación permite que los procesos cooperen entre sí en la ejecución de un objetivo global, en tanto que la sincronización permite que un proceso continúe su ejecución después de que un determinado evento ha ocurrido.

La comunicación entre tareas o procesos se puede realizar básicamente de dos maneras:
a) comunicación a través de un área común de memoria, o
b) comunicación mediante el intercambio de mensajes.

En el caso a) es necesario contar con mecanismos de sincronización para garantizar la consistencia de los datos almacenados como por ejemplo semáforos, monitores, etc.

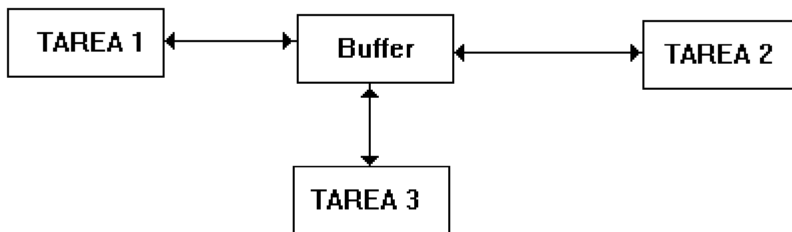


Fig. 19.2. - Comunicación a través de un área común de memoria.

En el caso b) las variables son locales. Un mensaje al llegar a su destino sólo es entregado a su tarea cuando ésta lo requiere. Si lo requiere y aún no está disponible la tarea debe suspenderse y esperar la llegada del mensaje.

Lógicamente los lenguajes necesarios para lograr estos objetivos deben:
- permitir multitareas,
- manejar sentencias del tipo SIGNAL, WAIT, DELAY, etc (es deseable),
- ser de tipo modular,
- proveer soporte para monitores, rendez-vous, etc.

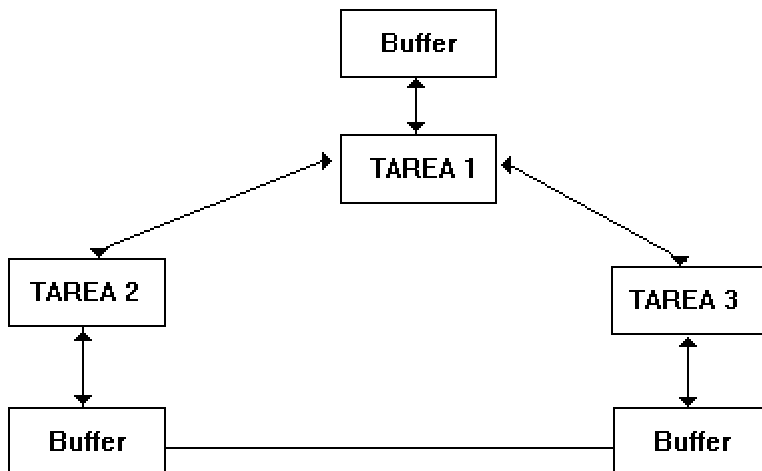


Fig. 19.3. - Comunicación mediante el intercambio de mensajes.

19.3. - **TAREAS (o Procesos)**

Un programa secuencial especifica una ejecución secuencial de una lista de instrucciones, siendo esta ejecución una tarea (o proceso).

Un programa concurrente especifica dos o más tareas (programas secuenciales) que pueden ser ejecutadas concurrentemente como tareas paralelas.

Un caso muy simple sería un programa listador de tarjetas: Tarea lectora, Tarea ejecutora, Tarea listadora.

Dividir un programa en tareas no significa simplemente dividirlo en programas menores, pero sí detectar y aislar las partes del programa que pueden ser potencialmente ejecutadas en paralelo.
Inclusive, no solamente se gana en estructura al dividirlo sino que se gana en eficiencia.

19.3.1. - Ejemplo

Programa

```
N = 40;
var
a: Vector (1..N) Integer;
k: Integer;
procedure Ordenar (inferior, superior);
    i, j, temp: Integer;
    begin
    for i := inferior to (superior - 1) do
        begin
        for j := (i + 1) to superior do
            begin
            if a(j) < a(i) then
                begin
                temp := a(j);
                a(j) := a(i);
                a(i) := temp
                end;
            end;
        end;
    end;
Begin (* programa principal *)
    for k:= 1 to n do Read(a,(k));
    Ordenar (1,n);
    for k:= 1 to n do Write (a(k));
end. (* programa principal *)
```

(Usaremos el número de comparaciones como una medida del tiempo de ejecución).

Examinando el algoritmo de ordenación (que no es el mejor pero es útil para ejemplificar) tenemos que el número total de comparaciones para ordenar n elementos es igual a:

$= (N-1) + (N-2) + ... + 1 = (N + N + ... + N) - (1 + ... + (N- 1)) =$

$= (N^2 - N) / 2 = (N (N - 1)) / 2$

que es aproximadamente igual a $N^2 / 2$.

Podemos disminuir el total de comparaciones si reemplazamos la sentencia Ordenar (1,N) por la siguiente secuencia:

```
Ordenar (1, N div 2);
    Ordenar (N div 2 + 1, N);
Combinar (1, N div 2 + 1, N);
```

en donde ordenamos por separado cada una de las mitades del vector y luego las combinamos en un único vector ordenado.

El ordenamiento involucra:

$2 * (N / 2)^2 / 2 = 2 * (N^2) / 8 + N$

Las últimas N comparaciones son necesarias para intercalar las dos mitades previamente ordenadas. Esto es aproximadamente igual a:

$(N^2 / 4) + N$ comparaciones.

Valiéndonos del hecho de que el ordenamiento de cada una de las mitades son actividades disjuntas, podemos ejecutarlas en paralelo en cuyo caso el algoritmo sería:

```
Com. Paralelo;
        Ordenar (1,x);
        Ordenar (x+1,n);
        Fin. Paralelo;
    Intercalar (1,x+1,n);
```

Y la cantidad de comparaciones se resume en:

$(N^2 / 8) + N$

La tabla 19.4 muestra el número de comparaciones necesarias para ordenar un vector de N elementos.

		Sin paralelismo	Con Paralelismo
n	$N^2 / 2$	$(n^2 / 4) + n$	$(n^2 / 6) + n$
40	800	440	240
100	5000	2.600	1350
1000	500.000	251.000	126.000

Tabla. 19.4.

19.4. - **Comunicación y Sincronización entre procesos**

Un programa concurrente puede ser ejecutado mediante tareas que compartan un único procesador o varios procesadores.

En el primer caso hablamos de multiprogramación. En el segundo hablamos de multiprocesamiento si es que todas las tareas comparten una memoria común. Si los procesadores están conectados por una red de comunicación estamos frente a un sistema distribuido.

El problema de tareas que ejecuten en forma concurrente es la independencia entre ellas, o sea, si sus variables son disjuntas pueden ejecutarse sin mayores inconvenientes, pero es muy común que tengan variables en común, e inclusive que se deban enviar mensajes entre sí para un perfecto funcionamiento (no intentar leer un buffer si éste no ha sido aún llenado, ni intentar llenarlo si todavía no ha sido leído, el ejemplo típico es el de los movimientos de actualización de los saldos de una cuenta corriente - depósitos previos a reembolsos, etc.-).

Además existe el problema del envío de los mensajes entre las tareas, la decisión de si el envío es sincrónico o asincrónico para seleccionar adecuadamente el mecanismo de comunicación.

Cuando las variables de una tarea son inaccesibles por otras tareas es fácil demostrar que el resultado final de la misma será una función independiente del tiempo.

Se dice en este caso que las tareas son disjuntas y que no existe **interferencia** entre las mismas. No sucede lo mismo cuando una tarea puede modificar las variables de otra tarea, pues en este caso, el resultado obtenido por esta última dependerá de las velocidades relativas entre las tareas.

Las tareas concurrentes que están involucradas en una aplicación global necesitan de mecanismos de comunicación y sincronización para lograr una cooperación efectiva entre los mismos.

Ejemplo de **interferencia** entre 2 procesos que usan una variable compartida.

P1	P2
a) LOAD X	d) LOAD X
b) ADD 1	e) ADD 2
c) STORE X	f) STORE X

Inicialmente X = 0; las posibles secuencias son:

a b c d e f ==> X = 3
a d b e c f ==> X = 3
a b d e c f ==> X = 2
d e a b c f ==> X = 1
....

resulta pues evidente la necesidad de establecer restricciones en la ejecución de las tareas concurrentes, o sea que es necesario sincronizarlas.

19.5.- GRAFOS DE PRECEDENCIA.

Supongamos un segmento de programa que lee desde 2 cintas (A y B) y escribe en una tercera (C).

```
- - - - -
Read (A, a);
Read (B, b);
c := a + b;
Write (C, c);
- - - - -
```

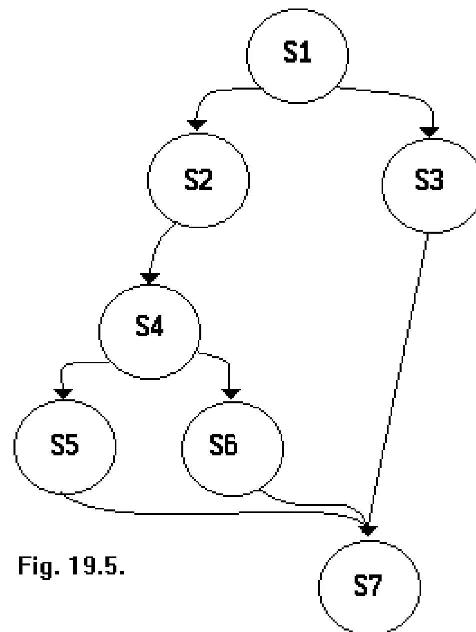


Fig. 19.5.

La variable c no puede ser grabada en la cinta C hasta que no se haya terminado de ejecutar la operación c:=a+b; y este cálculo no puede realizarse sin la ejecución previa de las dos lecturas. Sin embargo, es posible efectuar las lecturas concurrentemente, pues son independientes.

Las restricciones de precedencia entre las sentencias se pueden representar mediante el uso de un GRAFO DE PRECEDENCIA que NO tiene que tener ciclos.

En Fig. 19.5 Cada nodo indica una instrucción o un conjunto de instrucciones de ejecución secuencial.

El grafo de la Fig. 19.6 tiene ciclos, por lo tanto aquí no existe precedencia.

<u>DEFINICION</u> : Un grafo de precedencia es un grafo sin ciclos donde cada nodo representa una única sentencia. Un arco que parte del nodo S1 hacia el S2 indica que S2 puede ser ejecutado sólo si S1 ha completado su ejecución.

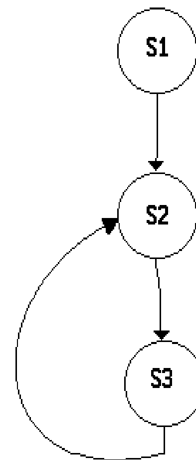


Fig. 19.6.

19.5.1. - <u>**CONDICIONES DE CONCURRENCIA.**</u>

En esta sección definiremos cuando dos o más instrucciones pueden ejecutarse concurrentemente. Previamente estableceremos cierta notación.

Conjunto lectura

R (Si) = (a1, a2, ... , am)

El conjunto lectura de la sentencia Si es aquel formado por todas las variables que son referenciadas por la sentencia Si durante su ejecución sin sufrir cambios.

Conjunto escritura

W (Si) = (b1, b2, ... , bn)

El conjunto escritura de la sentencia Si es aquel formado por todas las variables cuyos valores son modificados durante la ejecución de Si.

Ejemplo:

R (Read(a)) = (Ø) W (Read(a)) = (a)
R (Read(b)) = (Ø) W (Read(b)) = (b)
R (c := a + b)) = (a, b) W (c := a + b)) = (c)

- *<u>DEFINICION</u> : Dos sentencias cualesquiera Si y Sj pueden ejecutarse concurrentemente produciendo el mismo resultado que si se ejecutaran secuencialmente sí y sólo sí se cumplen las siguientes condiciones:*

1. R (Si) ∩ W (Sj) = (Ø).
2. W (Si) ∩ R (Sj) = (Ø).
3. W (Si) ∩ W (Sj) = (Ø).

Estas condiciones se conocen con el nombre de *Condiciones de Bernstein*. La idea es muy sencilla pero requiere de múltiples comparaciones.

19.5.2. - **Corrutinas**

Las corrutinas son similares a las subrutinas pero permiten una transferencia de control más flexible (Fig. 19.7).

La que realiza la transferencia de control es la instrucción RESUME.

Este mecanismo implementa la sincronización entre tareas para un único procesador, luego sirven para un ambiente de multiprogramación, y no para el caso de procesamiento en paralelo, ya que este sistema permite la ejecución de una rutina a la vez.

En SIMULA I, BLISS y MODULA-2 se implementaron comandos para este tipo de mecanismos.

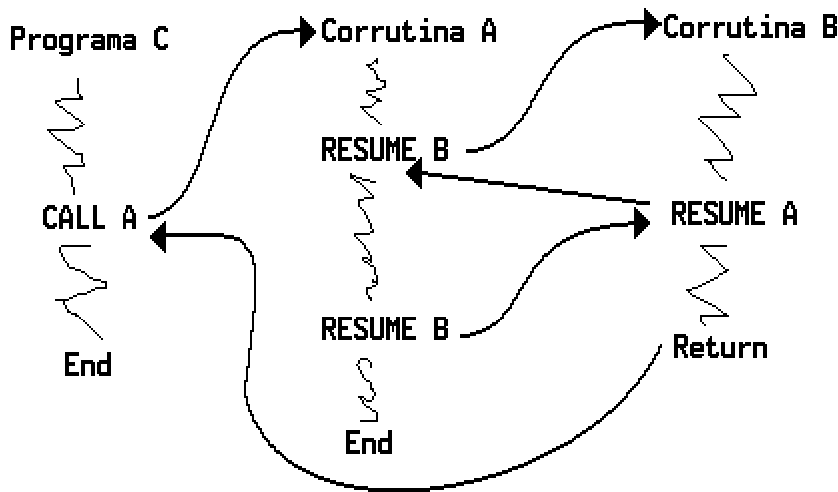


Fig. 19.7. - Esquema de funcionamiento de las Corrutinas.

La diferencia con las rutinas comunes estriba en que las corrutinas mantienen su estado anterior de ejecución, esto es, los resultados o modificaciones realizados durante la última invocación permanecen para la próxima vez que se la utilice (debido a que cuentan con una memoria local que permanece).

Además son sólo un mecanismo de control y no está previsto el pasaje de parámetros ni la comunicación.

19.6. - **INSTRUCCIONES FORK Y JOIN.**

Una forma sencilla de determinar la precedencia es que el mismo programador establezca las relaciones entre las instrucciones, estableciendo de esta forma cuando comienza la concurrencia (fork) y cuando termina (join).

La instrucción fork (tenedor, horqueta, separador) crea concurrencia y la instrucción join (junta) recombina concurrencia en una única secuencia.

Estas instrucciones fueron presentadas por Conway (1963) y por Dennis y Van Horn (1966).

La instrucción fork L1 produce dos ejecuciones concurrentes en un programa.

Una ejecución comienza a partir del rótulo L1, mientras que la otra prosigue con la ejecución de la sentencia que está a continuación de la instrucción fork.

Para ilustrar este concepto, consideremos el siguiente segmento de programa y su grafo de precedencia correspondiente :

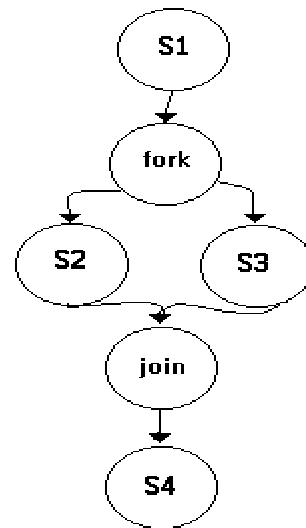


Fig. 19.8.

```
- - -
S1;
fork L1;
S2;
```

```
            - - -
            go to L2
L1:         S3;
L2:         join;
S4;
```

Cuando la sentencia fork L1 es ejecutada, un nuevo proceso es creado y comienza a ejecutar con la sentencia S3. El proceso original continua su ejecución con la sentencia S2.

La instrucción join brinda los medios para transformar a dos procesos concurrentes en un único proceso.

Es necesario que ambos procesos soliciten la combinación. Debido a que los procesos pueden ejecutar a velocidades diferentes estas solicitudes pueden ejecutarse en diferentes instantes. En este caso, el proceso que ejecuta el join en primer lugar es automáticamente terminado, mientras que al segundo proceso se le permite continuar.

Si fuesen tres los procesos a ser juntados o combinados, los dos primeros en ejecutar el join son terminados y sólo al tercero se le permite continuar.

La instrucción join tiene un parámetro que especifica el número de procesos a ser combinados. A tal efecto introducimos la variable count que dentro del join opera como:

```
        count = count - 1;
If count not = 0 then Espera;
```

Luego el ejemplo anterior puede traducirse como:

```
            S1;
            count := 2;
            fork L1;
            - - - - -
            S2;
            go to L2;
L1:         S3;
L2:         join count;
            S4;
            - - - - -
```

Ejemplos:

a) Consideremos el segmento de programa (el de la lectura de a y b) del párrafo 19.5.

Para permitir la ejecución concurrente de las dos primeras sentencias, este programa debería ser reescrito usando instrucciones fork y join.

```
            count := 2;
            fork L1;
            Read (a);
            go to L2;
L1:         Read (b);
L2:         join count;
            c := a + b;
            Write (c);
```

b) el ejemplo de la Fig. 19.5 puede traducirse como:

```
            S1;
            cuenta = 3;
            fork L1;
            S2;
            S4;
            fork L2;
            S5;
            go to L3;
L2 :        S6;
            go to L3;
L1:         S3;
L3:         join cuenta;
            S7;
```

c) El siguiente programa copia un archivo secuencial f en otro archivo llamado g. Debido a que usa dos buffers distintos para las operaciones de lectura y escritura puede efectuar ambas operaciones simultáneamente.

```
            Begin
            Read (f, r);
            while not eof(f) do
                    begin
                    count := 2;
```

```
                    s := r;
                    fork L1;
                    Write (g, s);
                    go to L2;
          L1:       Read (f, r);
          L2:       join count;
                    end;
          Write (g, r);
          end.
```

Las instrucciones fork y join constituyen un medio poderoso para escribir programas concurrentes. Desafortunadamente, los programas que usan estas sentencias tienen una estructura de control inadecuada debido a que la instrucción fork es en cierto sentido similar a un go to.

19.7. - INSTRUCCION DE CONCURRENCIA COBEGIN/COEND.

La sentencia cobegin/coend (o similarmente parbegin/parend) constituye una forma estructurada de especificar la ejecución concurrente de una o más tareas.

La ejecución **cobegin T1, T2, ... , Tn coend** implica la ejecución concurrente de T1, T2, ... , Tn correspondiente al grafo de precedencia de la Fig. 19.9.

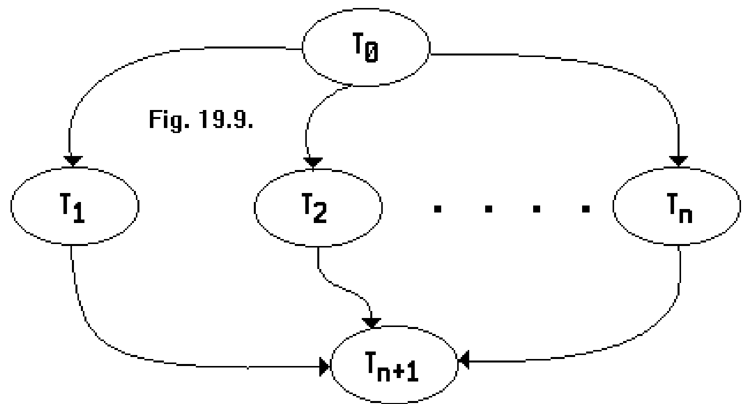


Fig. 19.9.

Cada uno de los Ti's puede ser cualquier comando, incluso un bloque cobegin/coend. La ejecución de un cobegin termina solamente cuando todos los Ti's han terminaron.

Ejemplos:

a) Codificación del segmento de programa del párrafo 19.5.(la lectura de a, b, y c=a+b).

```
     Cobegin
          Read (a);
          Read (b)
     Coend ;
     c := a + b;
     Write (c);
```

b) el ejemplo de la Fig. 19.5 puede traducirse como:

```
     S1
     Parbegin
          S3
          begin
               S2
               S4
               parbegin
               S5
               S6
               parend
          end
     parend
S7
```

c) Copia de un archivo en otro.

```
     begin
          Read (f, r);
          while not eof(f) do
               begin
                    s := r;
                    Cobegin
                    Write(g, s);
                    Read (f, r);
                    Coend
               end;
          Write (g, r);
     end.
```

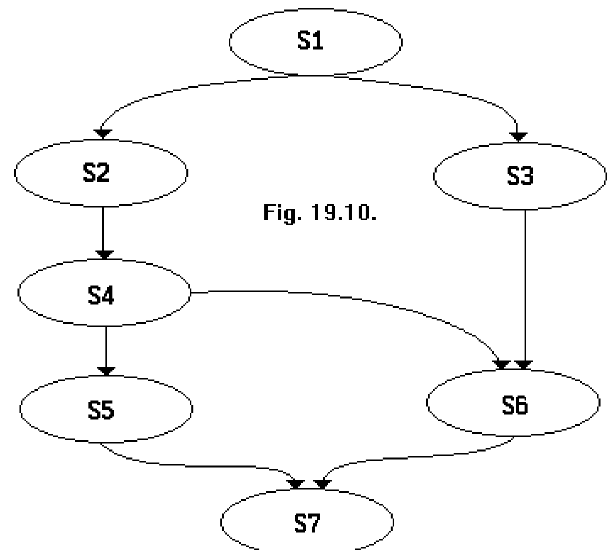


Fig. 19.10.

Podemos ver que esta instrucción de concurrencia es comparable a las estructuras modernas.

La instrucción de concurrencia cobegin/coend tiene como defecto el no poder representar a todos los grafos de precedencia.

Por ejemplo, sea el grafo de la Fig. 19.10.Este grafo puede ser codificado usando instrucciones fork y join aunque no por medio de las instrucciones parbegin/parend.

```
        S1;
        cuenta1 = 2;
        fork L1;
        S2;
        S4;
        cuenta2 = 2;
        fork L2;
        S5;
        go to L3;
L1:     S3;
L2:     join cuenta1;
        S6;
L3:     join cuenta2;
S7;
```

Si bien esta instrucción no cubre todos los grafos de precedencia, se estima que puede representar a todos aquellos que están relacionados a problemas reales.

19.7.1. - **Instrucción Cobegin/Coend expresada mediante Fork/Join**

Mostramos ahora cómo la instrucción de concurrencia se puede escribir mediante la instrucción fork/join.

Sea la instrucción:

PARBEGIN S1; S2; ....; SN PAREND

Se formula el siguiente segmento de proceso equivalente:

```
        cuenta = n;
        fork L2;
        fork L3;
        ...
        fork Ln;
        S1;
        go to Lj;
L2:     S2;
        go to Lj;
        ...
        ...
Ln:     Sn;
Lj:     join cuenta;
```

19.8. - **PROCESOS.**

19.8.1. - EL CONCEPTO DE PROCESO SECUENCIAL.

Un proceso secuencial es un programa en ejecución. Un programa por si solo no es un proceso; un programa es una entidad pasiva, mientras que un proceso es una entidad activa.

La ejecución de un proceso se realiza en forma secuencial. Esto significa que en cualquier instante de tiempo a lo sumo una instrucción del proceso es ejecutada. A pesar de que dos procesos pueden estar asociados con el mismo programa, se deben considerar como dos secuencias de ejecución disjuntas.

19.8.2. - ESTADOS DE UN PROCESO.

Un proceso secuencial puede estar en uno de los siguientes cuatro estados:

* Ejecutando: sus instrucciones están siendo ejecutadas por la CPU.
* Bloqueado: el proceso está espe-

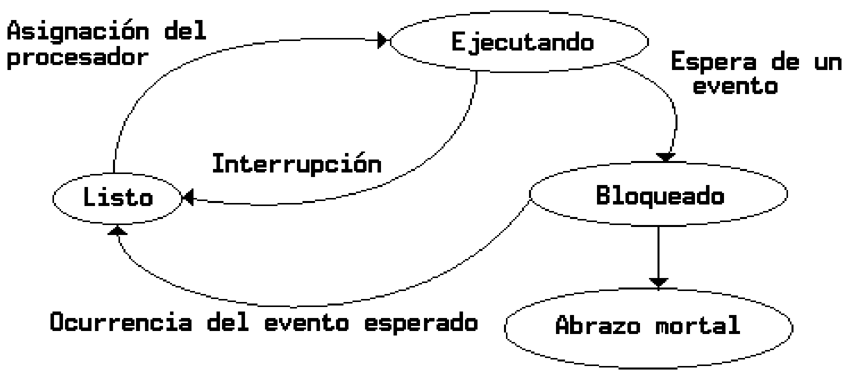


Fig. 19.11. - Diagrama de estados de un proceso.

rando la ocurrencia de algún evento (por ejemplo la terminación de una entrada/salida).
* Listo: el proceso está esperando que se le asigne un procesador.
* En abrazo mortal: el proceso está esperando por un evento que nunca ocurrirá.

19.8.3. - GRAFO DE PROCESOS.

Ahora veamos a cada nodo de un grafo de precedencia como un proceso secuencial. Como los procesos (tareas) aparecen y desaparecen crean mucho "overhead", luego es necesario que sean de ejecución secuencial, pero por supuesto teniendo cuidado de no disminuir la concurrencia.

Conviene, por lo tanto, reunir en un único proceso a todas aquellas instrucciones que pueden ser ejecutadas secuencialmente. El grafo resultante recibe el nombre de **grafo de procesos**.

La definición del grafo de procesos es muy similar al grafo de precedencia donde cada nodo es un proceso y

Fig. 19.12

significa que P_i crea al proceso P_j.

O sea que aquí no se habla de precedencia sino de Padre e Hijo. En precedencia P_j solo se podía ejecutar luego de P_i, y en el caso de grafos de procesos se dice que P_i **crea** a P_j.

19.8.3. - **OPERACIONES SOBRE PROCESOS.**

19.8.3.1. - **Creación de Procesos**

Un proceso puede ser CREADO y puede ser TERMINADO O ABORTADO, incluso por pedido propio. Aparte del proceso que se crea automáticamente al levantar el sistema, todo otro proceso es creado por un proceso que se denomina su PADRE.

Es posible que el padre esté activo en paralelo con sus hijos o que se suspenda hasta que éstos finalicen.

El primer caso se corresponde con el uso de instrucciones fork y join.

El segundo se cumple cuando se usan las sentencias cobegin y coend. En el caso de Parbegin S1,S2,....Sn Parend P_i crea estos n procesos concurrentes entre ellos y continúa su ejecución con la instrucción siguiente a la de la concurrencia luego de que estos finalicen.

19.8.3.2. - **Compartición de variables**

En este caso Padre/Hijo comparten todas sus variables (tanto Fork/Join como Parbegin/Parend).

Puede darse también que el hijo comparta sólo un subconjunto de las variables de su padre, como en el caso del Unix, P_j(hijo) tiene su memoria independiente de P_i(padre) y su comunicación sólo es posible por medio de archivos compartidos.

19.8.3.3. - **Terminación de procesos**

Un proceso termina cuando ejecuta su última sentencia o cuando otro proceso causa su terminación (terminación forzada, por ejemplo, por su padre))mediante el uso de un comando **Aborto (Kill) id**; donde **id** es el nombre del proceso a ser ABORTADO. La operación Kill sólo puede ser invocada por el proceso padre.

Para lograr esto el padre debería conocer todos los nombres de sus hijos lo que podría realizarse de la siguiente forma:
ID = Fork L

Muchos sistemas no permiten que un proceso hijo exista si su padre ha terminado. En un sistema con tales características si un proceso P termina entonces todos sus hijos son terminados a continuación.

Este fenómeno se conoce como *terminación en cascada* y es normalmente iniciado por el sistema operativo.

Las razones para terminar un proceso pueden ser:
- Exceso de uso de algún recurso
- La tarea asignada ya no es necesaria

19.8.3.4. - **Procesos estáticos y dinámicos**

Si un proceso no termina mientras el sistema operativo está funcionando se lo denomina estático. Si termina se lo denomina dinámico.

Hay sistemas operativos estáticos en donde todos sus procesos se crean en la medida de lo necesario pero no desaparecen luego. Su grafo es estático digamos que luego de un corto tiempo que sigue a la inicialización del sistema operativo.

El modelo de sistema sobre el cual trabajaremos será del tipo que puede visualizarse en la Fig. 19.13.

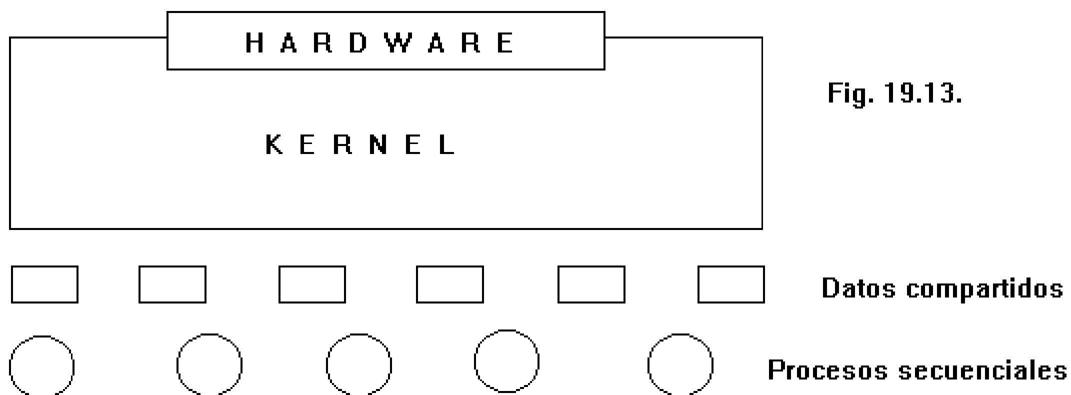


Fig. 19.13.

Las funciones del Kernel son:

a) Mecanismos de creación y eliminación de procesos.
b) Administración del procesador, memoria y dispositivos para estos procesos.
c) Herramientas de sincronización entre procesos.
d) Herramientas de comunicación entre procesos.

19.9. -**PROBLEMAS CRITICOS DE LA CONCURRENCIA**

La exclusión mutua es uno de los problemas más importantes en programación concurrente debido al hecho de ser la abstracción de muchos problemas de sincronización.

Veamos el problema del Productor/Consumidor con un buffer que es necesario vaciar hacia la impresora.

Si la cantidad de buffers fuese limitada, el productor pondría cada nueva información en un buffer nuevo y no habría problemas, aunque el consumidor podría adelantarse (luego debe esperar el productor).

Tomemos un número limitado de buffers (Fig. 19.14).

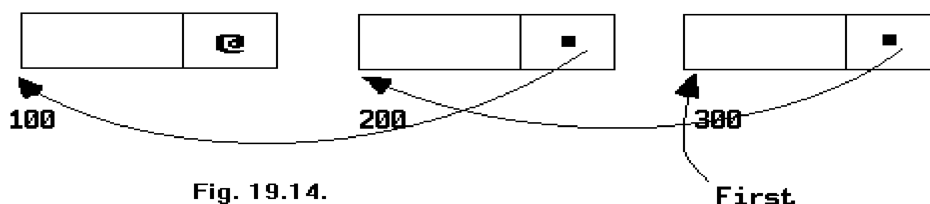


Fig. 19.14.

```
Buffer Inst : Dato
        Next : Apuntador
        P : Apuntador Buffer
        First : Apuntador = nil                (@)                    (Apunta al primero)
        Parbegin
                Productor :
                        Begin
                        Repeat
                        ...
                        Produce dato
                        ...
                        [400] (0)-------> New (P)
                        P.Inst = Dato
                        [300] (2)-------> P.Next = First
                        [400] (3)-------> First = P
                        End
                Consumidor :
                        Begin
                        While First = nil  do Skip ---->(Se queda ciclando)
                        [300] (1)-------> C = First
```

[300] (4)-------> First = First.Next
Dato = C.Inst — si C.Next (no apunta al mismo pero se perdió un buffer)
...
...
...
... — Consume dato
End

Parend

Si se realiza esa secuencia de instrucciones marcadas y con esos valores (los indicados entre corchetes) resulta :

- el buffer recién agregado se perdió
- First y C apuntan al mismo elemento

A esta situación se llegó debido a que se permitió que ambos procesos manipularan la lista simultáneamente.

Se dice que la actividad A1 de la tarea T1 y la actividad A2 de la tarea T2 deben excluirse si la ejecución de A1 no pueda ser intercalada con la ejecución de A2. Si T1 y T2 intentan ejecutar simultáneamente sus actividades Ai respectivas, se debe asegurar que sólo una de ellas proseguirá mientras que la otra permanecerá bloqueada hasta que la anterior termine la actividad Ai correspondiente.

Tales actividades Ai pueden ser, por ejemplo, leer o escribir variables comunes, modificar tablas, escribir sobre un archivo, etc.

Las secuencias de comandos Ai se conocen con el nombre de **regiones críticas**. Estas deben ser ejecutadas como operaciones indivisibles.

La idea es que, cuando un proceso esté ejecutando su "Sección Crítica", ningún otro lo pueda hacer, o sea que en ese punto todos los procesos sean mutuamente excluyentes en el tiempo.

Todo proceso antes de entrar a la región crítica debe "solicitar permiso". Esto es hecho en la "Sección de entrada". Además todo proceso cuando abandona la región crítica debe informar este hecho ejecutando la "Sección de salida".

El problema de exclusión mutua fue ampliamente tratado por Dijkstra, quien planteó las siguientes cuatro premisas:

1)- No deben hacerse suposiciones sobre las instrucciones de máquina ni la cantidad de procesadores. Sin embargo, se supone que las instrucciones de máquina (Load, Store, Test) son ejecutadas atómicamente, o sea que si son ejecutadas simultáneamente el resultado es equivalente a su ejecución secuencial en un orden desconocido.

2)- No deben hacerse suposiciones sobre la velocidad de los procesos.

3)- Cuando un proceso no está en su región crítica no debe impedir que los demás ingresen a su región crítica.

4)- La decisión de qué procesos entran a una parte crítica no puede ser pospuesta indefinidamente.

Los puntos 3) y 4) evitan bloqueos mutuos.

19.10. - **Algoritmos**.

Veamos algunos algoritmos de ejemplo.

Los procesos tendrán siempre la siguiente estructura o abstracción:

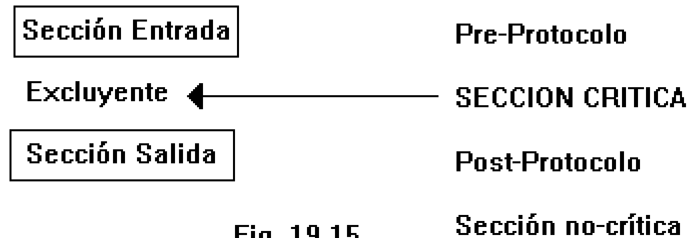


Fig. 19.15.

Si una tarea termina mal, no debe afectar a otras, luego los Protocolos también son regiones críticas.

19.10.1. - **Algoritmo 1**.

```
Sem inicializada en 0 ó 1 ó i
While Sem not = i  Do Skip
            SECCION CRITICA
      Sem = J
sección no-crítica
```

Esta solución asegura un proceso a la vez en la zona crítica pero no respeta la premisa 3), pues si Sem = 0 y el proceso P1 quiere hacer uso no puede aunque P0 no quiera usarlo.

Una tarea ejecuta en cadencia con otra (10 a 100 ?). Si no se ejecuta Sem = J las demás no entran.

19.10.2. - **Algoritmo 2**.

El problema anterior (lo hacemos para dos procesos) es que no se recuerda el estado de cada proceso, recordando sólo cual está queriendo entrar, para remediar esto reemplazamos la variable Sem por un vector donde cada elemento puede ser V ó F y está inicializado en F.

```
While Vector(j)  Do Skip
        Vector(i) = V
    SECCION CRITICA
    Vector(i) = F
    Sección no-crítica
```

Pero este algoritmo no asegura que un solo proceso esté en la zona crítica como puede verse si se sucede la siguiente secuencia:

```
T0    P0 encuentra Vector(1) = F
T1    P1 encuentra Vector(0) = F
T2    P1 pone Vector(1) = V
T3    P0 pone Vector(0) = V
```

La secuencia anterior puede ocurrir con varios procesadores o con uno solo y cuando el P0 sea interrumpido por un cualquier evento, por ejemplo, un reloj de intervalos.

19.10.3. - **Algoritmo 3**.

El problema anterior es que Pi toma una decisión sobre el estado de Pj antes que Pj tenga la oportunidad de cambiarlo; tratamos de corregir este problema haciendo que Pi ponga su indicador en V indicando que "solo" quiere "entrar" en la sección crítica.

```
Vector(i) = V
While Vector(j)  Do Skip
    SECCION CRITICA
Vector(i) = F
Sección no-crítica
```

Pero aquí la premisa 4) no se cumple, pues si se sucede: T0   P0 coloca Vector(0) = V

T1   P1 coloca Vector(1) = V

ambos procesos quedan en loop en la instrucción While (espera indefinida).

19.10.4. - **Algoritmo 4**.

La falla anterior se debió a no conocer el preciso estado en que se encontraban los otros procesos. Veamos ahora esta propuesta:

```
Vector(i) = V
While Vector(j)  Do
    Begin
    Vector(i) = F
    While Vector(i) Do Skip  (*)
        Vector(i) = V
    End
    SECCION CRITICA
    Vector(i) = F
```

Esta solución vuelve a no cumplir con la premisa 4), pues si ambos procesos ejecutan a la misma velocidad quedan en loop (Ver While (*)).

19.10.5. - **Algoritmo 5**.

Hasta ahora cada vez que encontrábamos un error y lo corregíamos aparecía otro, luego la solución no es trivial, aquí escribimos una solución correcta debida al matemático alemán T. Dekker.

Usa Vector(0,1) y Turno(0,1)

Inicialmente Vector(0) = Vector(1) = F y Turno = 0 ó 1.

```
Vector(i) = V
    While Vector(j) Do
        If Turno = j
        Then    Begin
```

```
                    Vector(i) = F
                    While Turno = j Do Skip (*)
                                        Vector(i) = V
                    End
          End do
SECCION CRITICA
Turno = j
Vector(i) = F
```

Con esto se puede verificar que :

a) existe la mutua exclusión ya que Pi modifica Vector(i) y solo controla Vector(j), y

b) el bloqueo no puede ocurrir porque Turno es modificado al final del proceso y habilita al otro.

En realidad Vector controla si se puede entrar y Turno cuándo se puede entrar. Si Turno es igual a 0 se le puede dar prioridad a P0.

Este algoritmo consiste en una combinación de la primera y la cuarta solución.

De la primera solución extrae la idea del pasaje explícito de control para entrar en la región crítica; y de la cuarta el hecho de que cada tarea posee su propia llave de acceso a la región crítica.

19.10.6. - **Algoritmo 6**.

El anterior algoritmo contempla el problema para 2 procesos, veamos la solución de Dijkstra (1965) para n procesos.

Usa Vector(0,......,n-1) que puede valer 0, 1 o 2; 0 indica ocioso, 1 indica quiere entrar y 2 indica dentro.

Inicialmente Vector(i) = 0 para todo i y Turno = 0 ó 1 ó .... ó n-1.

```
Repeat
          Vector(i) = 1;
          While Turno not = i Do
                    If Vector(Turno) = 0 then turno = i;
                    Vector(i) = 2;
                    j = 0;
                    While (j < n) and (j = i ó Vector(j) not = 2) Do j=j+1;
Until j > or = n;
SECCION CRITICA
Vector(i) = 0;
```

a) La mutua exclusión está dada porque solo Pi entra en su sección crítica si Vector(j) es distinto de 2 para todo j distinto de i y dado que solo Pi puede poner Vector(i) = 2 y solo Pi inspecciona Vector(j) mientras Vector(i) = 2.

b) El bloqueo mutuo no puede ocurrir porque:

- Pi ejecuta Vector(i) not = 0
- Vector(i) = 2 no implica Turno = i. Varios procesos pueden preguntar por el estado de Vector(Turno) simultáneamente y encuentran Vector(Turno) = 0. De todas maneras, cuando Pi pone Turno = i, ningún otro proceso Pk que no haya requerido ya Vector(Turno) estará habilitado para poner Turno=k, o sea no pueden poner Vector(k) = 2.

Supongamos {P1,.....,Pm} tengan Vector(i) = 2 y Turno = k ($1 \leq k \leq m$). Todos estos procesos salen del segundo While con j < n y retornan al principio de la instrucción Repeat, poniendo Vector(i) = 1, entonces todos los procesos (excepto Pk) entrarán en loop en el primer While, luego Pk encuentra Vector(j) not = 2 para todo i distinto de j y entra en su sección crítica.

Este algoritmo cumple con todo lo pedido, pero puede suceder que un proceso quede a la espera mientras otros entran y salen a gusto, para evitar esto se pone una quinta premisa.

5)- Debe existir un límite al número de veces que otros procesos están habilitados a entrar a secciones críticas después que un proceso haya hecho su pedido.

19.10.7. - **Algoritmo de Eisenberg y McGuire**.

El primer algoritmo que cumplió las premisas fue el de Knuth (1966) (esperando 2^n turnos), luego De Bruijin (1967) lo redujo a n^2 turnos y finalmente Eisenberg y McGuire (1972) desarrollaron uno que redujo la espera de turnos a n-1, y este es el algoritmo:

```
Repeat
          Flag(i) = Intento                    (quiere entrar)
          j = turno
          while j not = i  do
                    if flag(j) not = Ocioso then j = turno
                    else j = Mod n(j+1)
```

```
                    flag(i) = En =SC                        (en sección crítica)
                    j = 0
                    while (j < n) and (j=i or flag(j) not = En-SC) do j = j+1;
          Until (j ≥ n) and (Turno = i or flag(Turno) = Ocioso);
          SECCION CRITICA
          j = Mod n(Turno + 1)
          While (j not = Turno) and (flag(j) = Ocioso) do j = Mod n(j+1)
          Turno = j
          flag(i) = Ocioso
```

19.10.8. - **Algoritmo de la Panadería de Lamport**.

El principio es el de la entrega de un número de atención de clientes como en un negocio. Es un algoritmo que puede servir para sistemas distribuidos.

Este algoritmo no garantiza que a distintos procesos no se le entregue el mismo número, es este caso se atiende por orden de nombre.

Si Pi y Pj recibieron el mismo número y i < j se atiende primero a Pi (esta característica lo hace determinístico).

Usaremos la siguiente notación.

```
(a,b) < (c,d)      si a < c o si a= c es   b < d
Tomar.array(0,.....,n-1) boolean;          inicio F
Número.array(0,.....,n-1) integer;         inicio 0
```

Para Pi

```
Tomar(i) = V                                        (avisa que está eligiendo)
Número(i) = max (Número(0),...,Número(n-1)) + 1;             Aquí 2 procesos
                                                             pueden tomar el
                                                             mismo número
Tomar(i) = F
For j = 0 to n-1                                    (aquí comienza la secuencia
                                                     de control)
          Begin
          While Tomar(j) do Skip;                   (evita entrar en uno que
                                                             está eligiendo número
                                                             mientras es V)
          While Número(j) not = 0 and (Número(j),j) < (Número(i),i) do Skip;
                                                 (se asegura ser el menor de todos)
          end
SECCION CRITICA
Número(i) = 0;                                      (salida)
Sección no-crítica
```

La exclusión mutua está dada en el segundo While.

No puede ocurrir bloqueo mutuo pues en definitiva esto es un FIFO (FCFS).

19.11. **SOLUCIONES HARDWARE PARA LA EXCLUSION MUTUA**.

Muchas máquinas cuentan con instrucciones especiales de hardware que permiten testear y modificar el contenido de una palabra, o intercambiar el contenido de dos palabras en un único ciclo de memoria.

Estas instrucciones especiales pueden ser usadas para solucionar el problema de la exclusión mutua.

En lugar de tratar una instrucción específica para una máquina en particular, haremos abstracción de los conceptos principales implicados en este tipo de instrucciones definiendo la instrucción Test_and_Set de la siguiente forma:

Función Test_and_Set(x)

```
          begin
          Test_and_Set = x
          x = V
          end
```

Si la máquina cuenta con la instrucción Test_and_Set, la mutua exclusión puede ser implementada de la siguiente manera declarando previamente la variable booleana **lock** inicializada en FALSE.

```
     repeat
          while Test_and_Set (lock) do ;
          < SECCION CRITICA >;
          lock := FALSE;
```

```
                < SECCION NO CRITICA >
        forever
```
Por otra parte la instrucción Swap se define como :

```
procedure Swap (a,b)
                begin
                temp = a
                a = b
                b = temp
                end
```

Si la máquina cuenta con la instrucción Swap, la mutua exclusión puede ser implementada de una manera similar. Debe existir una variable global llamada **lock** inicializada en FALSE. Además cada proceso debe tener una variable local llamada **key**.

```
        key = V;
        repeat
                Swap (lock, key)
                until key = F
                < SECCION CRITICA >;
                lock = F
                < SECCION NO CRITICA >
        forever
```

La característica más importante es que estas instrucciones son ejecutadas atómicamente, en un único ciclo de memoria. De esta manera si dos instrucciones Test_and_Set (o Swap) son ejecutadas simultáneamente (cada una en una CPU diferente), ellas serán ejecutadas secuencialmente en algún orden arbitrario.

Veamos otra implementación del Swap:

```
COP    R1    R3    B2D2        Fig. 19.16.
       Op1   Op3   Op2
```

Si el primero y segundo operandos son iguales el tercero se intercambia con el segundo.
Si el primero y segundo operandos son distintos el segundo se intercambia con el primero.
Por ejemplo supongamos que:

B2D2 : es un semáforo que indica F (libre) o V (ocupado)
R1 : se testea su contenido luego del swap, si R1 = F ocupa y si R1 : V espera
R3 : V

Luego si se tiene:

Op1	Op3	Op2	
F	V	F	(Libre)
F	V	V	(Ocupa, ya que Op1 = F -libre- y pone al semáforo en V)

Por el contrario:

Op1	Op3	Op2	
F	V	V	(Ocupado)
V	V	V	(Ocupado, ya que Op1 = V)

19.12. - SEMAFOROS.

Los semáforos son banderas (señales) que indican la posibilidad de acceder o no a un recurso.
Las implementaciones hardware anteriores son ejemplos de semáforos, cuya forma general sería:
(V Ocupado, F Libre)

Cierre(x)	**Apertura(x)**
Begin	Begin
If x = V then Skip	x = F
else x = V	end
end	

Un caso particular son los semáforos contadores cuyo valor absoluto indica la cantidad de procesos que se encuentran en espera del recurso. Se manejan a través de los operadores P y V cuya estructura es la siguiente:

P(x)	**V(x)**
x = x - 1	x = x + 1
If x < 0 Wait(x)	if x $\leq$ 0 Signal(x)

Wait: primitiva que bloquea la tarea que ejecuta el Wait en la lista asociada al semáforo x.
Signal: primitiva que despierta una tarea que estaba bloqueada en una lista asociada a un semáforo.

Un ejemplo clásico de sincronización lo constituye:

Productor	**Consumidor**
P(E)	P(S)
....	....
V(S)	V(E)

con valores iniciales E = 1 y S = 0.

Los semáforos son una herramienta de uso general para resolver problemas de sincronización.

Los problemas de exclusión mutua son resueltos igual con ellos por medio de una operación P antes de entrar a la zona crítica y una V al salir.

Su implementación puede ser hardware o software y generalmente están incluidos en el núcleo de un sistema operativo.

```
X = 1
Procedure T1
    Begin
    P(x)
    CRIT
    V(x)
    end
Procedure T2
    Begin
    P(x)
    CRIT
    V(x)
    end
Cobegin
    T1; T2
Coend
```

Es de destacar la simetría y la simplicidad que se logra utilizando los operadores P y V, además se asegura la exclusión mutua y la ausencia de deadlock.

Para el caso de n tareas sería:

```
x = 1
Procedure Ti
    Begin
    P(x)
    CRIT
    V(x)
    end
Cobegin
    T1; T2;.........;Tn
coend
```

La sincronización condicional se realiza a través de una variable compartida que representa la condición y un semáforo asociado a la condición.

El caso más claro es el de procesos productores-consumidores:

```
Acceso-exclusivo = 1
Dato-depositado = 0
Procedure Tarea-Prod
    Begin
    Calculo(resultado)
    P(Acceso-exclusivo)
    Buffer = Resultado
    V(Dato-depositado)
    end

Procedure Tarea-Consum
    Begin
    P(Dato-depositado)
    x = Buffer
    V(Acceso-exclusivo)
    end
DO forever
Parbegin
    Tarea-Prod; Tarea-Consum
parend
```

En el ejemplo anterior las tareas son muy dependientes unas de otras, si se trabajase sobre un buffer de dimensiones adecuadas (en función de ambas tareas) el paralelismo aumentaría substancialmente pues mientras haya espacio en el buffer el productor podrá "escribir" y mientras haya datos el consumidor podrá "leer".

Un ejemplo es :

```
        Exclu = 1;
        Buff = Array(0,.....,Max-1);
        Vacío = Max;
        Ocupa = 0;
        Procedure Tarea-Prod
                Begin
                Calculo(Resultado)
                P(Exclu)                                (*)
                P(Vacío)                                (*)
                Buff(p) = Resultado
                p = (p + 1) mod Max
                V(Ocupa)                                (*)
                V(Exclu)                                (*)
                end
        Procedure Tarea-Consum
                Begin
                P(Exclu)                                (*)
                P(Ocupa)                                (*)
                x = Buff(c)
                c = (c + 1) mod Max
                V(Vacío)                                (*)
                V(Exclu)                                (*)
                end
        Begin Prog-Principal
                p = c = 0
                cobegin
                        Tarea-Prod
                        Tarea-Consum
                coend
        end
```

(*) Si se dejan en ese orden se bloquean pues se pide la exclusividad antes de saber si corresponde entrar. Se soluciona invirtiendo los P entre sí y los V entre sí.

La implementación de los semáforos se puede hacer en forma sencilla haciendo que la ejecución de un P o un V se transforme en una interrupción de software, y en caso (en el P) de estar ocupado "encolar" en una cola asociada al semáforo la tarea, cuando el recurso se libera (V) recorrer esa lista y despertar una tarea y enviarla a la cola de listos.

Otra implementación de semáforos (en monoprocesadores) es inhibir las interrupciones mientras se ejecutan los operadores P y V.

En sistemas de multiprocesamiento será necesario recurrir a soluciones software donde las secciones críticas son esencialmente los procedimientos P y V.

19.13. - LOS FILOSOFOS QUE CENAN.

Un grupo de cinco filósofos (encarnando cada uno a un proceso) conviven en un palacio donde su alimento es a base exclusiva de arroz, si bien su provisión es ilimitada.

Cada filósofo tiene su cuenco para comer, pero sólo cinco "palitos" (chopsticks) para todos.

Esto significa que cuando están sentados ante la mesa, que es circular, con sus cuencos delante de los "palitos" quedan de a uno entre dos filósofos.

La vida de un filósofo es un ciclo de pensar y comer. Otras necesidades de los filósofos son ignoradas en el problema.

El protocolo debe permitir a los filósofos comer el arroz de sus cuencos (obviamente, con dos "palitos") tomando y soltando los "palitos" de a uno.

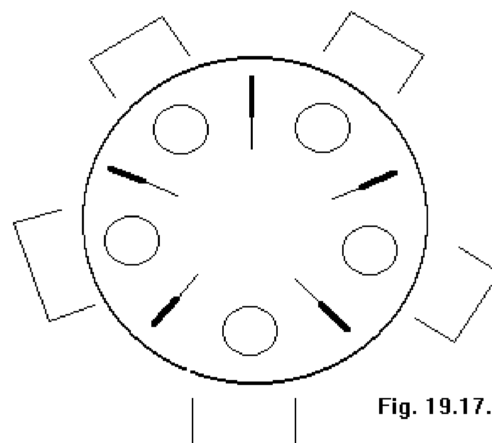


Fig. 19.17.

Se puede lograr una solución simple a este problema si asociamos a cada "palito" un semáforo. Un filósofo al intentar tomar un "palito" ejecuta una **P** sobre el semáforo correspondiente. Cuando

desea liberar un "palito" ejecuta una **V**.

```
Do forever
        P (palito (i));
        P (palito (mod 5 (i+1)));
        < Comer >;
        V (palito(i));
        V (palito (mod 5 (i+1)));
        < Pensar >
end;
```

A pesar de que este algoritmo garantiza de que dos filósofos vecinos no comen simultáneamente, no es correcto debido a que no previene el DEADLOCK y en consecuencia debería ser rechazado.

Supongamos que todos los filósofos sienten hambre al mismo tiempo y que cada uno toma el "palito" de la izquierda. Cuando intenten tomar el "palito" de la derecha quedarán bloqueados cada uno en un semáforo esperando a que dicho "palito" sea liberado. Como todos los filósofos asumen la misma postura, se producirá un abrazo mortal pues nadie libera los "palitos" sino hasta después de comer.

Algunas soluciones son :

- Permitir que los filósofos tomen los palitos si ambos están libres (esto sería la sección crítica)
- Los filósofos impares tomen primero el palito izquierdo y los filósofos pares tomen primero el palito derecho.

Aún así no se elimina el problema de la inanición que consiste en que ninguno decida tomar el palito.

19.14. - **MONITORES.**

La idea básica de este mecanismo es la de agrupar, en un mismo módulo, las regiones críticas relacionadas con un determinado recurso, haciendo que toda región crítica se encuentre asociada a un procedimiento del monitor, el que podrá ser llamado por las tareas.

La propiedad más importante que caracteriza a los monitores es que la ejecución de un procedimiento del monitor por una tarea excluye la ejecución de cualquier otro procedimiento del mismo monitor por otra tarea.

Un monitor se escribe como:

- un conjunto de declaración de variables
- un conjunto de procedimientos
- un cuerpo de comandos que son ejecutados inmediatamente después de la inicialización del programa que contiene al monitor.

La sintaxis de un monitor es la siguiente:

```
Monitor <nombre_del_monitor>;
        var < declaración de variables del monitor >
procedure <nombre_del_procedimiento1>;
                begin
                - - - - -
                end;
procedure <nombre_del_procedimiento2>;
                begin
                - - - - -
                end;
        begin
                < bloque de inicialización de variables del monitor >
        end.
```

Otras características básicas que merecen ser destacadas son las siguientes:

a) las variables declaradas en el monitor sólo pueden ser accedidas por los procedimientos del monitor;

b) la comunicación entre el monitor y el mundo exterior se realiza sólo mediante los parámetros de los procedimientos y

c) la única manera de ejecutar un monitor es a través de una llamada, de las tareas, a uno de los procedimientos definidos en él (es decir el monitor es pasivo).

Veremos el ejemplo del Productor-Consumidor.

Cuando el programa es iniciado el cuerpo del monitor es ejecutado primero y luego el programa principal dando inicio a la ejecución de las tareas concurrentes.

Para realizar sincronización los monitores disponen de dos primitivas: **Wait** y **Signal**, las que están asociadas a un nuevo tipo de variable llamada **variable de condición**.

Si una tarea ejecuta Wait(c), donde c es una variable de condición, es bloqueada en una cola de tareas asociadas a c y automáticamente se libera la exclusión mutua de la entrada al monitor.

Cuando se ejecuta un Signal(c), la primera tarea en la cola asociada a c es activada.

Como sólo una tarea puede estar ejecutando el monitor se debe establecer que cuando se ejecute un Signal(c), esa tarea abandone el monitor inmediatamente. Para esto diremos que en cada procedimiento existirá una sola instrucción Signal y que será el último comando del procedimiento.

En caso de que existan simultáneamente tareas en la cola asociada a c y tareas que quieran entrar al monitor por medio de una llamada normal, tienen prioridad aquellas que están bloqueadas en la cola asociada a c.

Ejemplos:

```
programa Productor_Consumidor;
MAX = ......;
monitor M;
buffer    : Array (0..MAX-1);
in, out, n; enteros;
buff_lleno, buff_vacío: condición;
procedure Almacenar (v);
        begin
        if n = MAX  then Wait (buff_vacío);
        buffer (in) = v;
        in = (in + 1) mod MAX;
        n = n + 1;
        Signal (buff_lleno)
        end;
procedure Retirar (v);
        begin
        if n = 0  then Wait (buff_lleno);
        v = buffer (out);
        out = (out + 1) mod MAX;
        n = n - 1;
        Signal (buff_vacio)
        end;
begin (* Cuerpo del monitor *)
        in, out, n = 0;
end;  (*     fin monitor    *)

procedure Productor;
        begin
        v = "dato producido"
        Almacenar (v)
        end;
procedure Consumidor;
        begin
        Retirar (v);
        Hacer algo con v
        end;
begin Programa-Principal
        Begin;
                cobegin
                Productor;
                Consumidor
                coend
        end.
```

Ejemplo: Simulación de semáforos

```
programa Exclusión-Mutua;
monitor Simulación de semáforo;
ocupado : boolean;
no-ocupado : condición;
procedure P (v);
        begin
        if ocupado  then Wait (no-ocupado);
        ocupado = V;
        end;
procedure V;
        begin
```

```
            ocupado = F;
            Signal (no-ocupado);
            end;
begin (* Cuerpo del monitor *)
            ocupado = F;
end;  (*     fin monitor    *)
tarea T1;
            begin
            P;
            Región Crítica;
            V;
            end;
tarea T2;
            begin
            P;
            Región Crítica;
            V;
            end;
Programa-Principal
            Begin;
                        cobegin
                        T1, T2;
                        coend
            end.
```

Comparando semáforos y monitores tenemos :
1)- Los monitores tienen garantizada la exclusión mutua.
2)- En los monitores la primitivas Wait y Signal son programadas internamente al monitor, o sea que el usuario del monitor debe solamente llamar a un procedimiento.
3)- Los semáforos deben ser explícitamente programados en el cuerpo de las tareas y por ende aumentan las probabilidades de errores.

19.15. - **NUCLEO PARA TIEMPO REAL**.

Las características que debe poseer un núcleo para tiempo real son :
- Atender eventos (interrupciones).
- Mantener más de una tarea simultáneamente en memoria.
- Permitir la interrupción de una tarea y reasumir su ejecución en el punto interrumpido.

Sus funciones son :
- Creación de tareas (bloque de control),
- Eliminación de tareas,
- Suspensión/Bloqueo de tareas,
- Reactivación de tareas,
- Cambio de prioridades de tareas,
- Scheduler de tareas,
- Adelantamiento o retardo de tareas (relacionado con el scheduling),

La creación de tareas requiere en este tipo de núcleos de:
- generar nombre
- dar prioridad
- armar el bloque de control
- alocación de los recursos necesarios para esa tarea.

19.16. - **PATH EXPRESSIONS**

Cuando se definen recursos que van a ser compartidos entre procesos concurrentes, se debe tener en cuenta la posibilidad de interferencias que dejen al recurso en un estado inconsistente.

Por ahora se ha visto que es posible evitar caer en interferencias por medio de los monitores que encapsulan al recurso y sólo permiten el uso del mismo a un proceso a la vez.

De todas maneras el monitor de por sí tiene un problema que surge de su definición (o sea un sólo proceso lo puede usar a la vez), pues hay operaciones que se pueden ejecutar simultáneamente sobre un recurso sin provocar interferencias.

Por ejemplo :

```
            Monitor Acceso
```

Proc-Lectura
....
Proc-Escritura
....
variables

Se podría permitir que alguno lea y otro escriba , sin embargo con el monitor esto no es posible.

Otro problema del monitor es que obliga la programación del recurso sin abstraerse del tema de la interferencia.

Una alternativa a esto son las Path Expressions que permiten la definición del recurso en dos partes.

- Definición normal de un tipo abstracto estableciendo el comportamiento del recurso, especificando operaciones y representación interna.
- Especificación del esquema de sincronización que deben seguir los procesos para acceder al recurso.

Ambas partes son conceptualmente independientes.

Con el monitor no se alcanza este nivel de modularidad, pues el esquema de sincronización del recurso compartido está esparcido en el interior de los distintos procedimientos de acceso al recurso.

Las Path Expressions fueron introducidas por Campbell y Habermann en 1974 y fueron flexibilizadas hasta usarse en una versión de Pascal Concurrente (Campbell/1979).

Luego las Path Expressions especifican el esquema de sincronismo en el acceso a un recurso compartido estableciendo un orden entre las operaciones de acceso al mismo, sin interferir con las operaciones en sí.

Su sintaxis es :

```
PE ::= PATH expr END
expr ::= sec
sec,expr
sec ::= elem
elem,sec
elem ::= entero : (expr)
[expr]
(expr)
id
```

siendo :
entero = entero sin signo
id = identificador de procedimiento

La semántica asociada es:

, indica concurrencia (ningún ordenamiento entre los operandos)
; indica secuencialidad entre los operandos
n : () indica a lo sumo n actividades concurrentes de lo que está entre paréntesis
[ ] indica ausencia de restricciones, no existe límite al número de activaciones concurrentes de lo que está entre corchetes

Veamos algunos ejemplos:

```
Path Inicio; [Escritura]; Fin END;
(escritura sobre una base de datos)
Path 10 : (Escribir; Leer) END;
(escribir  o leer en un buffer de 10 posiciones)
```

Las ventajas de las Path Expressions son :

- aproximación no procedural a la definición de sincronización de acceso a un recurso compartido
- mayor grado de modularidad
- mayor grado de paralelismo, pues no son tan restrictivas
- posibilidad de especificación de sincronismo únicamente en términos de secuencias de operaciones permitidas sobre el recurso

No son aptas para resolver problemas provenientes de una solución dependiente del estado de un recurso. En estos casos es necesario especificar una estrategia de acceso al recurso o reglas de prioridad de acceso (hay soluciones propuestas pero no son simples).

Para su implementación se necesitan lenguajes concurrentes y el Run-Time Support debe proveer las primitivas para su uso (por ejemplo semáforos).

Tomemos como ejemplo el caso del Productor-Consumidor.

```
Mailbox = object;
Path n : ( 1 : (Send); 1 : (Receive) ) end ;
Var buffer : array [0,......,n-1] de mensajes;
cabeza, cola : 0,......,n-1;
Procedure entry Send (x : mensaje);
        Begin
        buffer [cola] := x;
        cola := (cola + 1) mod n;
```

```
                end;
        Procedure entry Receive (var x : mensaje);
                Begin
                x := buffer [cabeza];
                cabeza := (cabeza + 1) mod n;
                end;
        Begin
                cabeza := cola := 0 ;
        End;
```

Los procedimientos Send y Receive son los más conocidos. La especificación de sincronización, definida por la Path Expression establece que :

a) muchas activaciones del procedimiento Send son, entre ellas, mutuamente excluyentes.
b) ídem con el procedimiento Receive.
c) cada activación de Receive es precedida por un Send .
d) el número de veces que se completa un Send no puede nunca superar n que es lo mismo (n) que se completa un Receive.

Estos 4 vínculos de sincronización nos aseguran la correcta ejecución del programa y que no pueda haber interferencias.

La diferencia con el monitor es que varias Send y Receive se pueden realizar concurrentemente.

Veamos posibles implementaciones, que nos permitan corroborar que se realiza la sincronización deseada.

Sea en primer término una sencilla :

```
Path 1 : (A,B) end ;
```

la primera aproximación es

```
P(x) - A, B - V(x)
```

con x inicializado en 1.

Ahora aquí existe concurrencia, por lo tanto es necesario proteger sus zonas críticas. lo que desemboca en :

```
P(x) - A - V(x)
P(x) - B - V(x)
```

o sea que en definitiva queda

```
        P(x)                P(x)
        cuerpo de A         cuerpo de B
        V(x)                V(x)
```

Sea ahora la implementación del Productor-Consumidor:

```
Path n : ( 1 : (Send) ; 1 : (Receive) ) end ;
```

obviamente esto se traduce en :

```
P(x) - 1 : (Send) ; 1 : (Receive) - V(x)     con x = n
```

y esto se traduce en :

```
P(x) - 1 : (Send) - V(y)
P(y) - 1 : (Receive) - V(x)                  con y = 0
```

luego como queremos que Send y Receive se ejecuten de a uno resulta :

```
        P(m)                      P(r)
        P(x)                      P(y)
        cuerpo del Send           cuerpo del Receive
        V(y)                      V(x)
        V(m)                      V(r)
```

con m = r = 1

19.17. - **PARADIGMAS DE PROGRAMACION**.

Como su nombre lo indica los Paradigmas de Programación son los diferentes modelos de programación. Existe cuatro grandes paradigmas :

- Algorítmicos
- Funcionales
- de Inferencia
- Orientado a Objetos

Los paradigmas de programación **algorítmicos** se dan en lenguajes tales como el Fortran o el Algol. Un ejemplo clásico es el siguiente programa (en Fortran):

```
        J = 1
        Do 100  I = 1, N
        J = I * J
100     Continue
```

De los paradigmas **funcionales** surgen lenguajes como el Lisp de tipo recursivo.

El equivalente a las instrucciones del caso de los algorítmicos son, en estos lenguajes, las **funciones** que se evalúan para obtener resultados.

```
F(x)
If x = 0  Then 1
          else x * F(x)
```

En los paradigmas de **inferencia** como ser en Prolog (Programming in Logic) no se escriben instrucciones interpretadas como hechos órdenes sino que se conforman de hechos lógicos y reglas de inferencia.

Hecho   Osvaldo ES-HIJO-DE José
Hecho   Carlos ES-HIJO-DE José
Regla   Si A ES-HIJO-DE B and C ES-HIJO-DE B entonces A ES-HERMANO-DE C

Luego, actúa un motor de inferencia que a partir de los hechos y las reglas infiere nuevos hechos.

Hecho inferido Osvaldo ES-HERMANO-DE Carlos

Aplicado a nuestro cálculo anterior se puede escribir:

```
F(1,1)<----------- indica 1 es 1!
F(x+1,y)<-------- F(x,z) & P(x+1,z,y)
```

donde y = (x+1)'.

Luego si se cumple

z = x! y se cumple y = z . (x+1)

En los paradigmas **orientados a objetos** se cuenta con objetos que poseen propiedades representadas por operaciones permitidas sobre esos objetos.

La idea es que todo el trabajo para construir el objeto se "resta" del trabajo necesario para construir el programa que lo utiliza.

Los objetos son encapsulados y sólo se puede acceder a ellos por medio de operaciones predefinidas (por ej.: los semáforos, siendo la forma de acceder a ellos las operaciones P y V).

Los objetos son accedidos a través de una capacidad. Se deben proveer los medios necesarios para poder utilizar y realizar las conexiones de los objetos entre sí.

19.18. - PROGRAMACION CONCURRENTE

Los Sistemas Operativos son escritos hoy en día en lenguajes de alto nivel.

Tradicionalmente los módulos de los sistemas operativos eran/son escritos en Assembler, pues los lenguajes de alto nivel:

- No preveían mecanismos para escribir código machine-dependent (como para manejo de dispositivos).
- No preveían herramientas apropiadas para escribir programas concurrentes.
- No producían código eficiente.

Los lenguajes actuales soportan concurrencia y son relativamente más eficientes.

Usar lenguajes de alto nivel permite un más fácil testeo, verificación, modificación y transportabilidad.

La ineficiencia de los códigos se resuelve utilizando técnicas de optimización.

Los lenguajes deben además proveer facilidades para la modularización y sincronización.

19.18.1. - Modularización

Es una técnica para dividir un programa largo en un conjunto de pequeños módulos. Trataremos las diferentes pautas para los Procesos, los Procedimientos y los Tipos de Datos Abstractos.

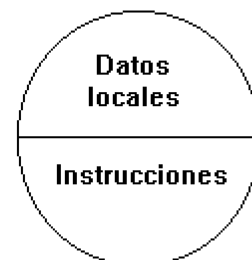


Fig. 19.18.

19.18.1.1. - Procesos

No deben compartir variables. Todas las variables que posean y utilicen <u>deben ser locales</u>.

Las variables en común deben ser definidas en un área común o encapsuladas en procedimientos o tipos abstractos de datos.

Deben mantener comunicación de acuerdo a lo visto anteriormente o pasando parámetros.

El compilador debería asegurar el <u>no</u> compartir variables.

19.18.1.2. - Procedimientos

Son subrutinas o funciones. Si es posible que tengan variables permanentes (que no desaparezcan entre distintas llamadas - por ej. corrutinas-) entonces es posible que los datos (variables) globales sean "encapsulados" en estos procedimientos.

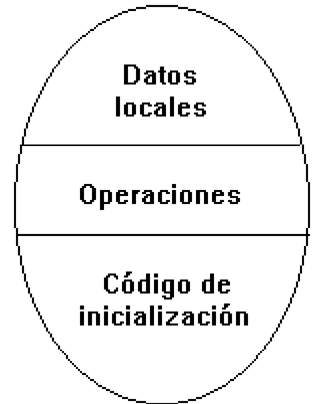


Fig. 19.19

19.18.1.3. - **Tipos de Datos Abstractos**.

Los procedimientos (funciones o subfunciones) son limitados pues si cambiamos el tipo de dato (entero a flotante, por ej.) será necesario determinar el error en ejecución (tiene un mecanismo limitado para esconder la información). El compilador determina el error del tipo de dato.

Se necesita, en consecuencia de un mecanismo que permita que un programador pueda crear una clase abstracta de objetos no definidos explícitamente (Tipos de Datos Abstractos).

Está caracterizada por :

- declaraciones de variables cuyos valores definen un estado de una instancia del tipo.
- conjunto de operadores (definidos por el programador)

El ejemplo clásico es el stack.

Este tipo de datos en general no viene como los enteros, flotantes, etc., lo debe definir el programador y a su vez tiene operaciones bien definidas que le permiten acceder a la información (push, pop, empty, top, etc.), y finalmente, sólo puede acceder a esa información con esos operadores y no directamente.

Los llamamos "Class" (Simula 67 - Pascal Concurrente).

La sintaxis de "class" es :

```
type class-name = class
(declaración de variables)
Procedure P1
Begin.........end
Procedure P2
Begin.........end
.......
Begin
        inicialización de código
end
```

No es necesario que todos los procedimientos sean vistos (exports), sólo los necesarios que marcamos con

```
Procedure Entry p(...);              (Sólo se pueden invocar los Entry)
```

Ejemplo: Administración de bloques libres (en memoria o disco) con mapa de bits.

```
Type Bloque = class;
var Libre : Vector [1.......n] of boolean;
Procedure Entry Ocupar (var Indice : entero);
        Begin
        for Indice = 1 to n
        do If Libre (Indice)
        then    Begin
                Libre (Indice) = F;
                exit;
                end;
        Indice = -1;
        end
Procedure Entry Liberar (Indice : entero);
        Begin
        Libre (Indice) = V;
        end
Begin
        for Indice = 1 to n
        do Libre (Indice) = V;
end
```

Luego es posible declarar

```
var memoria : Bloque;
```

y usar "memoria.Ocupar(n);" y "memoria.Liberar(n);".

Las ventajas son que se pueda cambiar la implementación del bit map, por ejemplo por una lista encadenada.

A nivel de compilación es posible detectar errores que de otra forma sólo aparecerían en momento de ejecución.

19.18.2. - **Sincronización**

Si dos procesos hacen uso simultáneo de "Bloque" (la rutina anterior) tendremos problemas de inconsistencias.
Es necesario que el compilador tenga herramientas apropiadas de sincronización.

19.18.2.1 - **Regiones Críticas**.

Los semáforos son una herramienta adecuada. Pero deben ser usados en forma apropiada, por ejemplo:

```
Si      V(x)
        ...
        P(x)
```

causa no-exclusión, entonces

```
        P(x)
        ...
        P(x)
```

causa deadlock.
Para cuidar esto Hansen y Hoare (1972) crearon un nuevo tipo de variable (tipo Región Crítica).
Una variable v de tipo T y que será compartida, se declara como :

```
var v : shared T ;
```

y cuando se la utilice

```
region v do S;
```

o sea si el conjunto S utiliza la variable v lo podrá hacer únicamente de la forma anterior y de tal manera que si dos procesos la quieren utilizar en forma concurrente

```
region v do S1;
region v do S2;
```

los mismos serán ejecutados en algún orden secuencial.
Una implementación posible de esto cuando el compilador se encuentra con

```
var v : shared T
```

es que genere un semáforo x = 1 y cuando se encuentra con

```
region v do S;
```

genere

```
P(x)
S;
V(x)
```

También se puede tener un anidamiento de regiones críticas:

```
var x,y : shared T
        Parbegin
        Q1 : region x do region y do S1;
        Q2 : region y do region x do S2;
        parend
```

En este caso nótese que hemos construido un deadlock:

```
T0      Q1 P(x-x)            (léase x de x)
T1      Q2 P(y-x)
T2      Q2 P(x-x)            luego espera
T3      Q1 P(y-x)            luego espera
```

Luego el compilador debería detectar estas situaciones o sino generar un orden. por ejemplo si y está en x e (y< x), reordenarlo.

19.18.2.2. - **Regiones Críticas Condicionales** (Hoare 1972).

```
region V When B Do S;
```

donde B es una expresión booleana, que es evaluada, y si es verdad entonces se ejecuta S.
Si B es falsa el proceso libera la exclusión mutua y espera hasta que B sea verdadera y no exista otro proceso ejecutando en la región asociada a v.
Veamos un ejemplo recreando el caso del productor/consumidor.
El buffer usado está encapsulado de la siguiente forma:

```
var buff : shared record
b : array [ 0.....n ]
count, in, out, enteros
```

```
        end
```
El proceso Productor inserta un nuevo elemento PEPE:

```
        region buff when count < n
                do begin
                b(in) = PEPE;
                in = (in+1) mod n
                count = count + 1;
                end
```

El proceso Consumidor retira en SPEPE:

```
        region buff when count > 0
                do begin
                SPEPE = b(out);
                out = (out+1) mod n;
                count = count - 1;
                end;
```

El compilador debe trabajar de esta forma cuando se encuentra con una declaración. Por cada variable x compartida debe tener las siguientes variables asociadas:

```
        var  x-sem, x-wait : semáforos;
        x-count, x-temp : enteros;
```

Donde :

x-sem = controla la exclusividad en la región crítica.
x-wait = es el semáforo por el que hay que esperar cuando no se cumple la condición.
x-count = cuenta la cantidad de procesos esperando por x-wait.
x-temp = cuenta las veces que un proceso testeó la condición (booleana)

Los valores iniciales son :

x-sem = 1
x-wait = 0
x-count = 0
x-temp = 0

El compilador frente a la region x when B do S podría implementarlo de la siguiente manera :

```
        P(x-sem)
                If not B
                Then    begin
                        x-count = x-count + 1;
                        V(x-sem);               libera exclusión
                        P(x-wait);              se pone en espera
                        while not B             se despierta pero no se cumple aún condición
                        do      begin
                                x-temp = x-temp + 1;
                                if x-temp < x-count  (el sólo?)
                                        then V(x-wait)
                                        else V(x-sem);
                                P(x-wait);
                                end;
                        x-count = x-count - 1;
                        end;
        S;
        if x-count > 0
                then    begin
                        x-temp = 0
                        V(x-wait)
                        end;
        else V(x-sem);
```

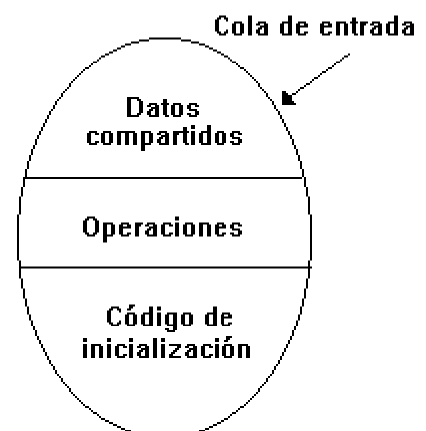


Fig. 19.20.

El caso anterior tiene problemas cuando son muchos los procesos, pues todos ellos deben testear su condición por lo cual se debe esperar que ello ocurra.

Otra construcción (Hansen) supone que el testeo de la condición puede realizarse en cualquier punto de la región:

```
        region v
                do Begin
                S1;
                AWait (B);
                S2;
        end
```

Todos estos ejemplos tienen el problema que cada "procedimiento" tiene que proveer su propio mecanismo de sincronización en forma explícita.

19.18.2.3. - **Monitores**.

Para salvar el problema anterior Hansen (1973) y Hoare (1974) desarrollaron una nueva construcción del lenguaje : el Monitor.

El mecanismo monitor permite un seguro y efectivo compartir "tipos de datos abstractos" entre varios procesos.

La sintaxis es igual que en "class", pero reemplazada por "monitor".

La diferencia semántica principal es que el monitor asegura la mutua exclusión (por definición), o sea, sólo un proceso puede estar activo dentro del monitor.

Luego el programador no necesita explicitar sincronización.

Igual que antes se necesitan ayudas para lograr algún grado de sincronización, cosa que se obtiene con las construcciones "condición".

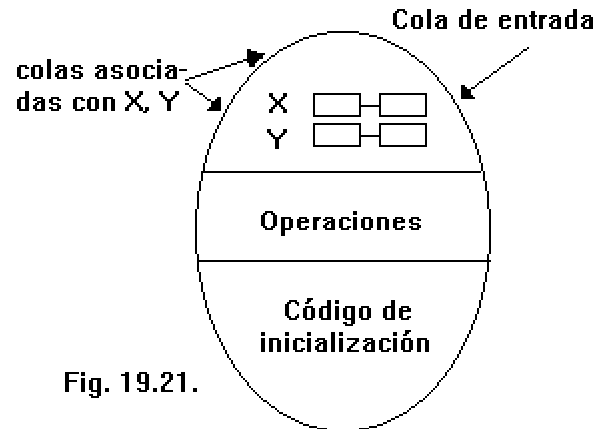


Fig. 19.21.

Por ejemplo : Var x, y : condición;

Las únicas operaciones permitidas son x.wait; y x.signal;

Cuando P ejecuta un signal pueden darse dos opciones:

a) P espera por ejecución de un Q (Hoare 1974)

b) Q espera hasta que P termina (Hansen 1975)

Si dejamos seguir P puede suceder que la condición esperada por Q pase de largo (es decir que no sea más necesaria su ejecución).

Utilicemos de ejemplo un semáforo binario:

```
type semáforo = monitor;
var ocupado : boolean;
nocupado : condición;
Procedure Entry P;
        Begin
        If ocupado then nocupado.wait;
        ocupado = V;
        end
Procedure Entry V;
        Begin
        ocupado = F
        nocupado.signal
        end
Begin
        ocupado = F
end
```

Veamos ahora una solución de Monitores para el problema de los filósofos que cenan:

```
type Filósofos = Monitor;
var estado : array [ 0,...,4 ] of (Pensar, Hambre, Comer);
var ocio : array [ 0,...,4 ] of condición;   (para quedar en espera si <R>
tiene hambre pero no puede comer)
Procedure Tomar-Palitos (i : 0,...,4 )
        Begin
        estado(i) = Hambre;
        test (i);
        if estado(i) not = Comer then ocio(i).wait;
        end;
Procedure Dejar-Palitos ( i : 0,...,4 );
        Begin
        estado(i) = Pensar;
        test ( (i-1) mod 5 );
        test ( (i+1) mod 5 );
        end
Procedure test ( k : 0,...,4 )
        Begin
        if estado ( (k-1) mod 5 ) not = Comer and
        estado (k) = Hambre and
        estado ( (k+1) mod 5 ) not = Comer          (*)
```

```
                    then    begin
                            estado (k) = Comer;
                            ocio(k).signal;
                            end;
                    end
            Begin
                    for i = 0 to 4
                    do estado(i) = Pensar;
            end
```

La forma de invocarlo sería:

```
x.Tomar-Palitos;
......
Comer
......
x.Dejar-Palitos;
```

Esta solución nos asegura que dos vecinos no coman simultáneamente y no ocurra abrazo mortal. Se debería ver también qué pasa si uno de los filósofos se muere lo cual puede preguntarse en (*).

Una implementación sencilla de monitores sería la siguiente:

```
P(x)
Procedimiento
V(x)
```

Si queremos que los que están en espera tengan prioridad se puede realizar :

```
P(x)
Procedimiento
If cuenta > 0              (cuenta los que están en espera)
        then V(próximo)
else V(x)
```

Valores iniciales x=1, Próximo = 0, Cuenta = 0 (x.sem = 0 asociado a la variable x).

La implementación del x.wait puede ser

```
if cuenta > 0
        then V(próximo)
        else V(x);
P(x-sem)
```

La implementación de x.signal puede ser :

```
Begin
        cuenta = cuenta + 1;
        V(x-sem);
        P(próximo);
        cuenta = cuenta - 1;
end
```

19.18.3. - **Pascal Concurrente**.

Este lenguaje soporta modularidad en la construcción de programas.

Maneja tres tipos de módulos:

- Procesos : son secuenciales y no comparten variables.
- Clases : tipos abstractos de datos (cada clase puede ser accedida por un solo proceso).
- Monitores : soporte para la comunicación de procesos
  - variables condicionales: tienen una sola entrada (se llaman queues).
  - signal : el que lo emite abandona inmediatamente el monitor.

Estas restricciones permiten armar eficientes programas para sistemas operativos.

Este lenguaje no permite procedimientos recursivos ni tampoco tipos de datos recursivos.

Toda la memoria es estática en el momento de la compilación.

Todas las instancias (componentes del programa) son creadas por declaración y son permanentes.

19.18.4. - **CSP - Communicating Sequential Processes**.

Este lenguaje está orientado a programación concurrente para redes de microcomputadoras con memoria distribuida (Hoare 1978).

Los conceptos centrales del lenguaje son :

- El programa CSP consiste de un número fijo de tareas (procesos secuenciales) mutuamente disjunto en sus espacios de direcciones.
- La comunicación y sincronización están dadas a través de construcciones input/output.

- Las estructuras de control secuenciales están basadas en las precauciones (guarded) de Dijkstra.

19.18.4.1. - **Comunicación**.

La comunicación ocurre cuando un proceso nombra a un segundo como destino y el segundo al primero como fuente (similar al Send/Receive).

El mensaje es copiado del primero al segundo.

La transferencia tiene lugar cuando ambos invocan los comandos (output, input). Los procesos pueden ser suspendidos hasta que el otro haga la operación recíproca.

Este mecanismo sirve de comunicación y sincronización.

La notación sintáctica no es convencional.

Productor
Consumidor ! m (significa Send m to Consumidor)
Consumidor
Productor ? n (significa Receive n from Productor)

El tipo de m y n debe ser el mismo, caso contrario la comunicación no se produce y ambos procesos quedarán en wait (por siempre).

Si uno de los procesos termina, subsiguientes invocaciones a él producirán terminaciones anormales.

19.18.4.2. - **Estructuras de Control Secuenciales (notación Dijkstra)**.

<guarda> ------> <lista de comandos>

Guarda: lista de declaraciones, expresiones booleanas y un comando de input (cada uno es opcional).

Una guarda falla si alguna de sus expresiones booleanas falla (es falsa) o si los procesos nombrados en sus comandos de input han terminados.

Si una guarda falla, el proceso que la contiene aborta.

Si no falla entonces la lista de comandos se ejecuta (esto ocurre sólo después que el comando de input -si está presente- ha sido completado).

Las guardas pueden ser combinadas en comandos alternativos.

Por ejemplo:

[G1 --> C1 □ G2 --> C2 □ ..... □ Gn --> Cn]

Esto especifica la ejecución de una de las guardas. Si todas las guardas fallan el comando alternativo falla y aborta el proceso.

Si más de una guarda puede ser ejecutada se elige una "arbitrariamente".

Los comandos alternativos pueden ejecutarse en forma **repetitiva** por ejemplo:

*[G1 --> C1 □ ..... □ Gn --> Cn]

El comando alternativo puede ejecutarse todas las veces que sea posible. Cuando todas sus guardas fallan, falla él también y la transferencia se pasa a la siguiente instrucción.

Ejemplo:

```
Consumidor/Productor Buffer de 10.
buffer : ( 0,....9 );
in = 0;
out = 0;
*       [ in < out + 10; productor ? buffer ( (in) mod 10 ) ----> in = in + 1;<R>
        □ out < in; consumidor ? more () -->     consumidor ! buffer ( (out mod 10 );
                                                 out = out + 1; ]
```

El productor lo usaría como :

```
Pepe-buffer ! p;
```

El consumidor como :

```
Pepe-buffer ! more ();
Pepe-buffer ? q;
```

19.18.5. - **ADA**.

Para programas concurrentes este lenguaje se basa en el concepto de tarea.

Las tareas se comunican y sincronizan a través de:

- Accept : combinación del Call y transferencia.
- Select : estructura determinística basada en las guardas de Dijkstra.

Accept <entry-name> [<lista de parámetros>] [do <instrucciones> end;]

Esto se puede ejecutar sólo si otra tarea invoca el entry-name (o sea hace Call). También se pasan los parámetros.

Cuando se llega al end los parámetros son devueltos y ambas tareas quedan libres.

Cualquiera de las dos tareas pueden ser suspendidas hasta que la otra esté lista (luego sirve de comunicación y sincronización).

La selección entre varios entry-call se realiza con el Select, que tiene la siguiente forma aproximada :

```
Select
[ when < boolean > ==> ]
        <accept>
        [ <instrucciones> ]
{ or [ when < boolean > ==> ]
        <accept>
        [ <instrucciones> ]
[ else < instrucciones > ]
end Select;
```

(El caracter { indica repetición).

Donde:

1)- Todas las boolean son evaluadas. Cada accept que tenga su boolean en V es marcada "open". Un accept sin when es siempre marcado open.

2)- Un accept "open" sólo puede ser seleccionada para ejecución si otra tarea la invocó. Si hay varias se hace elección arbitraria. Si ninguna puede ser seleccionada se ejecuta el else. Si no hay else entonces la tarea queda en Wait.

3)- Si ninguna puede ser seleccionada y no hay else se produce una condición de excepción.

El Accept provee el mecanismo para que una tarea por eventos predeterminados de otra.

El Select provee el mecanismo de esperar un conjunto de eventos cuyo orden no puede ser predicho.

Ejemplo: Productor/Consumidor.

```
Task Body Pepe-buffer is
buffer : array [ 0,...,9 ]
in = 0;
out = 0;
cuenta = 0;
Begin
        Select
        when cuenta < 10 ===>
                Accept inserción (ítem)
                do buffer [ mod 10 (in) ] = ítem end;
                in = in + 1;
                cuenta = cuenta + 1;
        or when cuenta > 0 ===>
                Accept sacar (ítem)
                do ítem = buffer [ mod 10 (out) ] end ;
                out = out + 1;
                cuenta = cuenta -1;
        end Select;
end
```

El productor realizará Pepe-buffer.inserción (p);

El consumidor hará Pepe-buffer.sacar (q);

CAPITULO 20

SISTEMAS DISTRIBUIDOS.

20.1. - **Evolución de Arquitectura de Computadoras**.

Repasemos ahora nuevamente los conceptos de arquitecturas paralelas pero desde otra perspectiva.

El estudio de la arquitectura de computadoras involucra tanto la organización del hardware como los requerimientos de programación/software. Un programador de lenguaje ensamblador ve la arquitectura del computador como una abstracción del conjunto de instrucciones, incluyendo los códigos de operación, los modos de direccionamiento, registros, la memoria virtual, etc.

Desde el punto de vista hardware la máquina abstracta está organizada con CPUs, caches, buses, microcódigo, pipeline, memoria física, etc. Sin embargo, el estudio de la arquitectura cubre tanto la organización del conjunto de instrucciones como la de la implementación de la máquina.

En las pasadas cuatro décadas la arquitectura de las computadoras ha sufrido cambios evolutivos más que revolucionarios. Se comienza con la arquitectura Von Neumann construida como una máquina secuencial ejecutando datos escalares. La computadora secuencial fue mejorada desde la realización de operaciones de bits en forma seriada a operaciones de palabras en paralelo, y desde operaciones en punto fijo a operaciones en punto flotante. La arquitectura Von Neumann es lenta debido a la ejecución secuencial de las instrucciones del programa.

20.2. - **Lookahead, Paralelismo y Pipelining**.

Las técnicas de lookahead fueron introducidas para poder superponer las operaciones de I/E (fetch de instrucciones/ decodificación y ejecución) y habilitar el paralelismo funcional. El paralelismo funcional se logró de dos formas: una es utilizando múltiples unidades funcionales simultáneamente, y la otra es utilizar pipelining en varios niveles de procesamiento.

Esta última incluye ejecución de instrucciones pipelinizadas, cálculos aritméticos pipelinizados, y operaciones de acceso a memoria.

20.3. - **Clasificación de Flynn** (1972).

Hemos visto con anterioridad la clasificación de Flynn basada en la cantidad de flujos de datos y de instrucción que procesa un computador.

20.4. - **Computadoras paralelas/vectoriales**.

Las computadoras intrínsecamente paralelas son aquellas que procesan programas en modo MIMD. Existen dos grandes clases de estas computadoras, a saber, las *computadoras de memoria compartida* y las *computadoras con pasaje de mensajes*. La mayor distinción entre multiprocesadores y multicomputadoras reside en la compartición de la memoria y en los mecanismos de comunicación entre los procesos.

20.5. - **Atributos de un sistema para la performance**

20.5.1. - **Tasa de Reloj y CPI**

La CPU de las computadoras digitales de hoy en día está gobernada por un reloj con un *tiempo de ciclo* constante (τ en nanosegundos). La inversa de este tiempo es la *tasa de reloj* ($f = 1 / \tau$) en megahertz).

El tamaño de un programa esta determinado por la *cantidad de instrucciones* (I_c). Las instrucciones de diferentes máquinas pueden requerir diferentes ciclos de reloj para ejecutarse. Sin embargo, los *ciclos por instrucción* (CPI) es un importante parámetro para medir el tiempo necesario para ejecutar dicha instrucción.

20.5.2. - **Factores de performance**.

Sea I_c como lo definimos antes. El tiempo de CPU (T en segundos/programa) necesario para ejecutar el programa se estima hallando el producto :

$$T = I_c * \text{CPI} * \tau \qquad (1)$$

Definimos un ciclo de memoria como el tiempo necesario para completar una referencia a memoria. Usualmente un ciclo de memoria es *k* veces el ciclo de procesador τ. El valor de *k* depende de la velocidad de la memoria y del esquema de interconexión de memoria-procesador.

El CPI de una instrucción tipo puede dividirse en dos componentes correspondientes al total de ciclos del procesador y de ciclos de memoria necesarios para completar la ejecución de la instrucción. Dependiendo del tipo

de instrucción el ciclo completo de la misma puede involucrar de uno a cuatro referencias a memoria (una para el fetch de la instrucción, dos para los operandos, y otra para almacenar el resultado). Reescribimos entonces la fórmula (1) de la siguiente forma :

$$I_c * (p + m * k) * \tau \qquad (2)$$

donde *p* es la cantidad de ciclos de procesador necesarios para decodificar y ejecutar la instrucción, *m* es la cantidad de referencias de memoria necesarias y *k* es la relación entre el ciclo de memoria y el ciclo de procesador.

20.6. - **Atributos de Sistema**.

Los cinco factores de performance anteriores están influidos por atributos del sistema tales como la arquitectura del conjunto de instrucciones, la tecnología del compilador, el control y la implementación de la CPU y la jerarquía de la memoria y de la cache de acuerdo a como se muestra en la siguiente tabla:

ATRIBUTOS DEL SISTEMA	**FACTORES DE PERFORMANCE**				
	Cantidad de instrucciones Ic	Promedio de Ciclos pos instrucción, CPI			Tiempo de ciclo de procesador, τ
		Ciclos de Procesador por instrucción, *p*	Referencias a Memoria por instrucción, *m*	Latencia de acceso a Memoria, *k*	
Arquitectura del set de instrucciones	**X**	**X**			
Tecnología del compilador	**X**	**X**	**X**		
Control e implementación del Procesador		**X**			**X**
Jerarquía de Memoria y cache				**X**	**X**

20.6.1. - **Tasa MIPS**.

Sea *C* el número total de ciclos de reloj necesarios para ejecutar un programa dado. Entonces el tiempo de CPU en (2) puede estimarse como

$$T = C * \tau = C / f.$$

Más aún,

$$CPI = C / I_c$$

y $$T = I_c * CPI * \tau = I_c * CPI / f.$$

La velocidad del procesador se mide a menudo en *millones de instrucciones por segundo* (MIPS),

$$\text{tasa MIPS} = I_c / (T * 10^6) = f / (CPI * 10^6) = f * I_c / (C * 10^6) \qquad (3)$$

Basándonos en esta ecuación se puede decir que el tiempo de CPU en la ecuación (2) también puede escribirse como

$$T = (I_c * 10^{-6}) / MIPS$$

Se concluye que los MIPS de un procesador son directamente proporcionales a la velocidad del reloj e inversamente proporcionales al CPI.

20.6.2. - **Tasa Throughput**.

Otra medida importante está relacionada con cuántos programas por unidad de tiempo un sistema puede ejecutar, esto se llama *throughput del sistema* W_s (en programas/segundo). En un sistema multiprogramado el throughput del sistema es a menudo inferior al *throughput del procesador* W_p al que definimos como:

$$W_p = f / (I_c * CPI) \qquad (4)$$

Nótese que de la ecuación (3) $W_p = (MIPS * 10^6) / I_c$. El throughput del procesador es una medida de cuántos programas pueden ejecutarse por segundo basándose en la tasa MIPS y el promedio de longitud del programa (I_c). La razón del porqué $W_s << W_p$ se debe a los overheads adicionales del sistema causados por las E/S, el compilador y el sistema operativo cuando varios programas se intercalan para su ejecución en un ambiente multiprogramado.

20.6.3. - **Ejemplo**

Considérese la utilización de una VAX/780 y una IBM RS/6000 para ejecutar un cierto programa de benchmark. Las características de las máquinas y la performance declarada se muestran en la siguiente tabla:

Máquina	Reloj	Performance	Tiempo de CPU
VAX 11/780	5 MHz	1 MIPS	12x segundos
IBM RS/6000	25 MHz	18 MIPS	x segundos

Estos datos indican que el tiempo de CPU medido para la VAX es 12 veces mayor que el medido para la RS/6000. Los códigos objeto corridos en ambas máquinas difieren en su longitud debido a la máquina y al compilador utilizado. Los otros overhead se ignoran.

Basándose en la ecuación (3) la cantidad de instrucciones del código objeto en la RS/6000 es 1,5 veces mayor que el código que corre en la VAX. Más aún, el promedio CPI en la VAX/780 se asume que es de 5 en tanto que en la RS/6000 es 1,39 ejecutando el mismo programa de benchmark.

No podemos calcular el throughput del procesador W_p a menos que conozcamos la longitud del programa y el promedio CPI de cada código.

20.7. - MULTIPROCESADORES Y MULTICOMPUTADORAS

Existen dos categorías de computadores paralelos. Estos modelos físicos se diferencian en el hecho de tener memoria compartida o distribuida.

20.7.1. - Multiprocesadores de memoria compartida.

Existen tres modelos que se diferencian en como la memoria y los periféricos se comparten o distribuyen, a saber:

- **UMA** (uniform memory access)
- **NUMA** (no uniform memory access)
- **COMA** (cache-only memory access)

20.7.2. - El modelo UMA

La memoria se comparte uniformemente entre los procesadores. Todos tienen igual tiempo de acceso a todas las palabras de memoria. Cada procesador puede tener una cache privada. Los periféricos también se comparten de la misma forma. Se los denomina *sistemas fuertemente acoplados*. Para coordinar los eventos paralelos, la sincronización e intercomunicación entre procesos se utilizan variables en la memoria común.

Cuando todos los procesadores tienen igual acceso a todos los periféricos el sistema se denomina *simétrico* (MP).

En un multiprocesador *asimétrico* solo uno o un subconjunto de los procesadores tienen la capacidad ejecutiva. El procesador ejecutivo o maestro ejecuta el sistema operativo y maneja las E/S. Los otros procesadores no pueden manejar las E/S y se los llama *attached processors (APs)*. Los procesadores attached ejecutan código bajo la supervisión del procesador maestro.

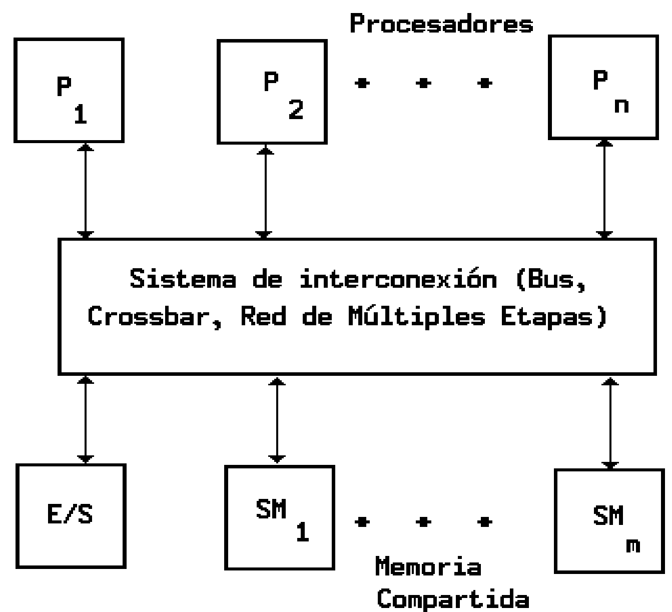


Fig. 20.1. - El modelo UMA. Sequent S-81.

20.7.3. - El modelo NUMA

Es un sistema de memoria compartida en donde el tiempo de acceso a memoria varía dependiendo de la ubicación de la palabra de memoria.

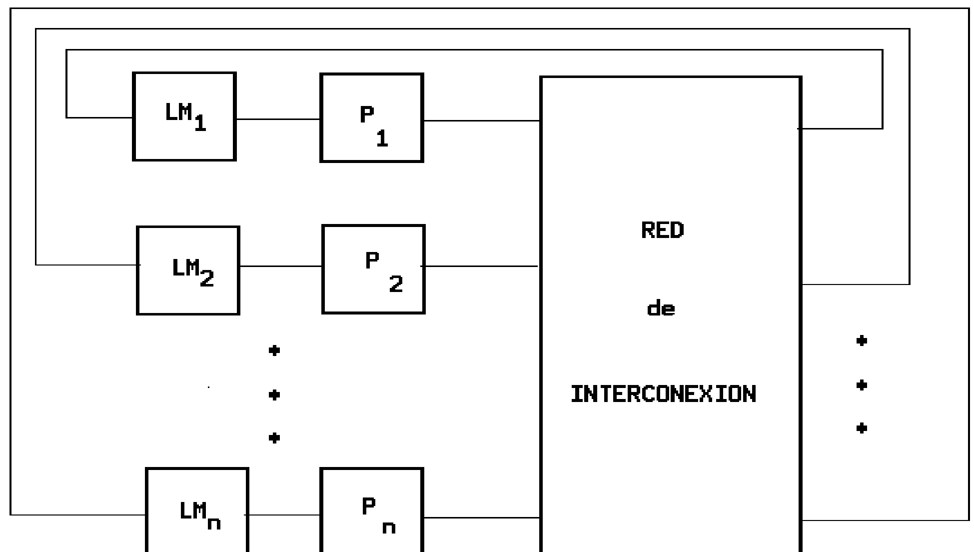


Fig. 20.2. - Memoria locales compartidas BBN Butterfly.

La memoria está físicamente distribuida entre todos los procesadores, se llaman *memorias locales*. Es más rápido acceder un dato en la memoria local, para acceder información en una memoria remota existe una demora debida a la red de interconexión (ej. la BBN TC-2000 Butterfly).

Además de estas memorias distribuidas se puede agregar memoria globalmente accesible por todos los procesadores. En este caso existen tres patrones de acceso a memoria:

- el más rápido es acceder la memoria local,
- un poco más lento es acceder la memoria global,
- y por último el más lento de todos es acceder la memoria remota de otro procesador (ej. Cedar de la universidad de Illinois)

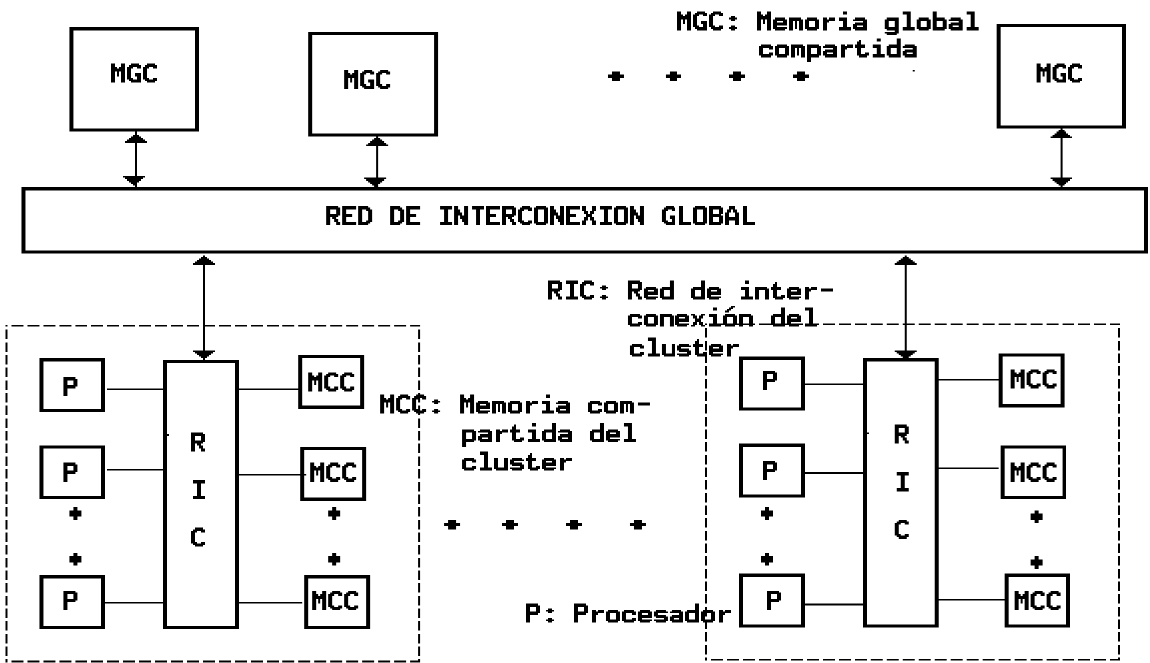


Fig. 20.3. - Cluster jerárquico (la Cedar).

20.7.4. - **El modelo COMA**

El modelo COMA utiliza solo memorias de tipo cache. (ej. la KSR-1 de Kendall Square Research). Son un caso particular de la NUMA en el cual la memoria distribuida se convierte en memorias cache. No existen jerarquías de memoria en cada nodo procesador. Todas las caches forman un espacio de direccionamiento global. El acceso a una cache remota se facilita a través de los directorios distribuidos de cache los cuales en ciertas arquitecturas pueden estructurarse en forma jerárquica.

Una de las mayores falencias que poseen estos multiprocesadores es la falta de escalabilidad debido a la centralización de la memoria compartida. Al construir sistemas MPP se logra mayor escalabilidad pero son menos programables debido a la adición de los protocolos de comunicación.

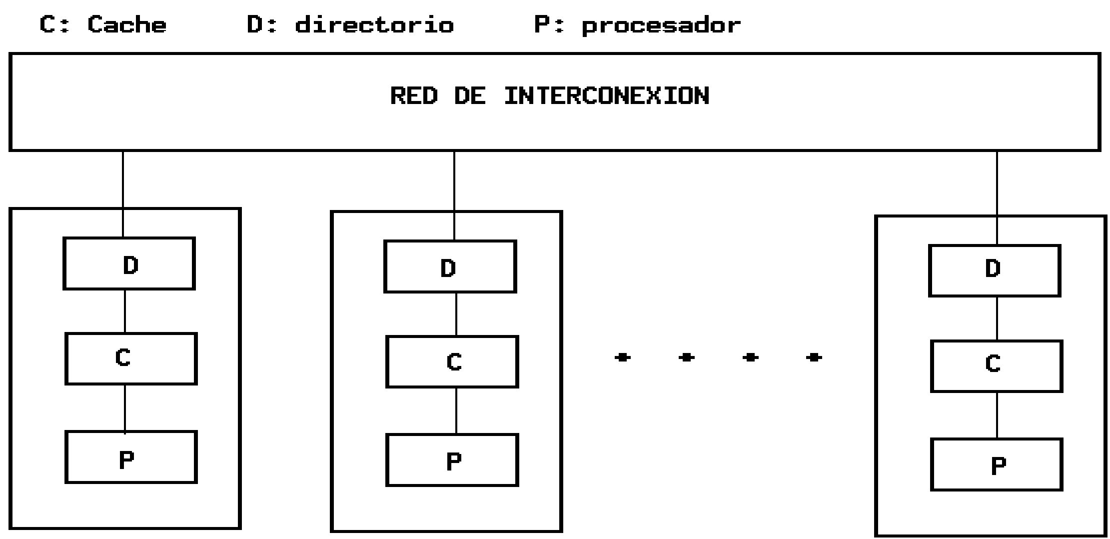


Fig. 20.4. - El modelo de multiprocesador COMA. La KSR-1.

20.8. - **Multiprocesadores de Memoria Distribuida**

Consisten de múltiples computadores llamados a menudo *nodos* interconectados por una red de pasaje de mensajes. Cada nodo es un sistema computador autónomo con su procesador, su memoria y a veces periféricos de E/S.

La red de pasaje de mensajes provee un mecanismo de comunicación punto-a-punto. Todas las memorias locales son privadas y son solo accesibles por el conjunto de procesadores locales del nodo, por esta razón se las denomina máquinas **NORMA** (no remote memory access). La comunicación entre los nodos se logra a través de la red de pasaje de mensajes.

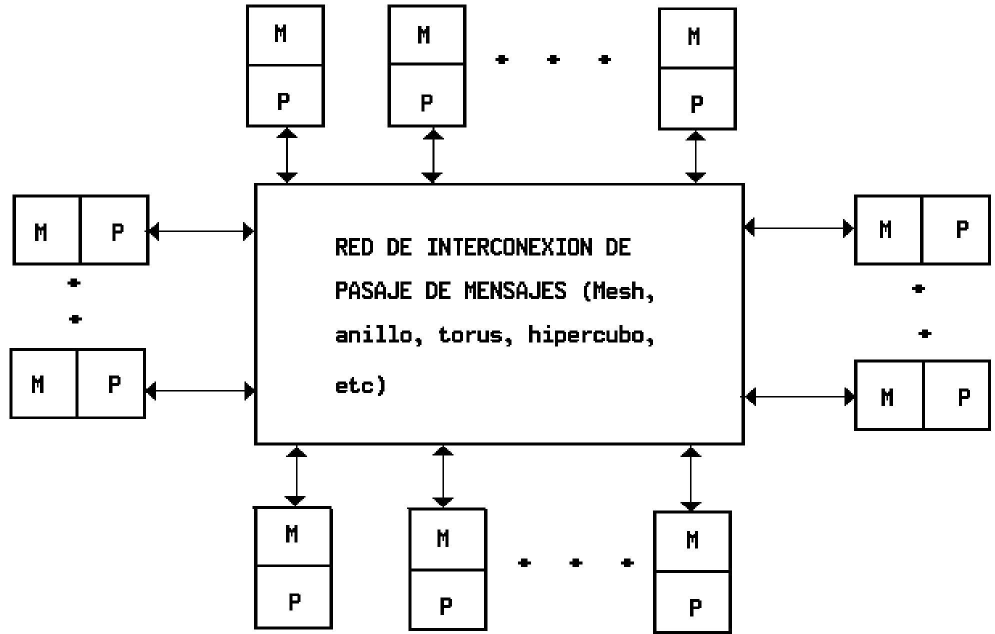


Fig. 20.5. - Multicomputador genérico. Pasaje de mensajes.

20.9. - **Generaciones de Multicomputadoras**

Las multicomputadores de pasaje de mensajes provienen de dos generaciones de desarrollo y existe una tercer generación emergente.

La *primera generación* (1983-1987) se basó en tecnología de la placa del procesador (board) utilizando arquitectura hipercubo y control de mensajes por software. Ejemplos de esta generación son la Caltech Cosmic y la Intel iPSC/1.

La *segunda generación* (1988-1992) se implementó sobre arquitecturas mesh-connected, ruteo de mensajes por hardware y un entorno software para distribuir medianamente el cómputo. Ejemplos de esta generación son la Intel Paragon y la Parsys SuperNode 1000.

La *tercera generación* (1993-1997) se espera que provea multicomputadoras de un refinamiento mayor tales como la MIT J-Machine y la Caltech Mosaic implementadas con mecanismos de transmisión de comunicación y los procesadores en el mismo chip VLSI.

20.10. - **UNA TAXONOMÍA DE COMPUTADORAS MIMD**

Las computadoras paralelas aparecen tanto como configuraciones SIMD y MIMD. Las SIMD parecen más apropiadas para aplicaciones específicas.

La tendencia arquitectural para las futuras computadoras de propósito general apunta a las MIMD con memorias distribuidas proveyendo un espacio de direccionamiento globalmente compartido.

Gordon Bell (1992) proveyó una taxonomía de máquinas MIMD que se muestra en la siguiente tabla.

Este autor considera los multiprocesadores de memoria compartida como que poseen un único espacio de direccionamiento.

Las multicomputadoras o multiprocesadores escalables deben utilizar memoria compartida distribuida.

Los multiprocesadores no escalables utilizan una memoria compartida central.

Las multicomputadoras utilizan memorias distribuidas con múltiples espacios de direccionamiento. Son escalables con memoria distribuida.

Las multicomputadoras centralizadas aún no han aparecido.

MIMD	**Multiprocesadores único espacio de direccionamiento Memoria compartida**	Multiprocesadores de memoria distribuida (escalables)	Procesadores con enlace dinámico de direcciones (KSR)
			Procesadores con enlace estático de direcciones, multi-anillo (estándar propuesto IEEE SCI)

			Procesadores con enlace estático de direcciones con uso de cache (Alliant, DASH)
			Enlace estático de programas (BBN, Cedar, CM*)
	Multicomputadores Múltiples espacios de direccionamiento Pasaje de mensajes	Multiprocesadores de memoria centralizada (no escalables)	Cross-point o múltiples etapas (Cray, Fujitsu, Hitachi, IBM, NEC, Tera)
			Multianillo, multibus
			Multibus (DEC, Encore, NCR, Sequent, SGI, Sun)
		Multicomputadores distribuidos (escalables)	Conexión Mesh (Intel)
			Butterfly / Árbol ancho (CM5)
			Hipercubo (NCUBE)
			LANs rápidas para alta disponibilidad y clusters de alta capacidad (DEC, Tandem)
			LANs para procesamiento distribuido (workstations, PCs)
		Multicomputadores centralizados	

20.11. - **INTRODUCCION A LOS SISTEMAS DISTRIBUIDOS**

20.11.1. - **Introducción**

Desde 1945 hasta el 85 las computadoras eran grandes y caras incluso las minicomputadoras. Sin embargo a partir de la década de los 80 hubo dos importantes avances tecnológicos:

- el desarrollo de poderosos microprocesadores (similares a los de los mainframes)
- el desarrollo de las redes de área local de alta velocidad (LAN)

Estos sistemas permitieron conectar varios computadores transmitiendo información entre ellos a muy alta velocidad. El resultado neto de estas dos tecnologías recibe el nombre genérico de Sistemas Distribuidos en contraste de los Sistemas Centralizados.

20.11.2. - **Ventajas de los S. D. con respecto de los Sistemas Centralizados**

Economía : Los microprocesadores ofrecen mejor proporción de precio/rendimiento que los mainframe.

Velocidad : Un Sistema Distribuido puede tener un mayor poder de cómputo que un mainframe ya que al agregar otra terminal, se incrementa dicha potencia mientras que en el mainframe hay que cambiar el procesador por otro más veloz, y esto no siempre es posible.

Distribución inherente : Algunas aplicaciones poseen una distribución inherente a sí mismas, por ejemplo las diferentes sucursales de un supermercado operan casi totalmente en base a operaciones y decisiones locales y necesitan comunicarse entre sí solo en lo que respecta a intercambio de información.

Confiabilidad : Si una máquina falla, el sistema sobrevive en un gran porcentaje.

Crecimiento incremental : Se puede añadir poder de cómputo en pequeños incrementos agregando más computadoras.

20.11.3. - **Ventajas de los S. D. con respecto a las PC.**

Datos compartidos : Permite el acceso por varios usuarios a una base de datos en común. Facilita la coherencia y consistencia de la información.

Dispositivos compartidos : Permiten que varios usuarios compartan periféricos caros como ser impresoras láser, etc. Inclusive remotamente.

Comunicación : Facilita la comunicación de persona a persona. Por ejemplo correo electrónico.

Flexibilidad : Distribuye la carga de trabajo entre las máquinas disponibles en forma más eficaz en cuanto a los costos.

20.11.4. - **Desventajas**

Software : Existe poco software para sistemas distribuidos en la actualidad.
Redes : La red se puede saturar o causar otros problemas como ser pérdida de mensajes, ruido en la línea, etc.
Seguridad : Si bien el poder compartir los datos es una ventaja se torna también en un problema la privacidad de los mismos.

20.12. - CONCEPTOS DE HARDWARE

La clasificación de hardware más citada es la de Flynn que si bien se detiene en los tipos SISD, SIMD, MISD y MIMD, para un sistema distribuido, se usa la arquitectura MIMD, dividiéndola en aquellas que tiene memoria compartida, usualmente denominadas *Multiprocesadores* (Sistemas fuertemente acoplados) y aquellas que no, llamada *Multicomputadoras* (Sistemas débilmente acoplados).

La diferencia fundamental es que en los sistemas multiprocesadores existe un espacio de direcciones virtuales compartido por todas las CPU y en la multicomputadoras cada máquina tiene su propia memoria.

Cada una de estas categorías se pueden subdividir en base a la arquitectura de la red de interconexión: *bus* o *conmutador* (switch).

Los sistemas fuertemente acoplados tienden a ser usados como sistemas paralelos y los débilmente acoplados como sistemas distribuidos.

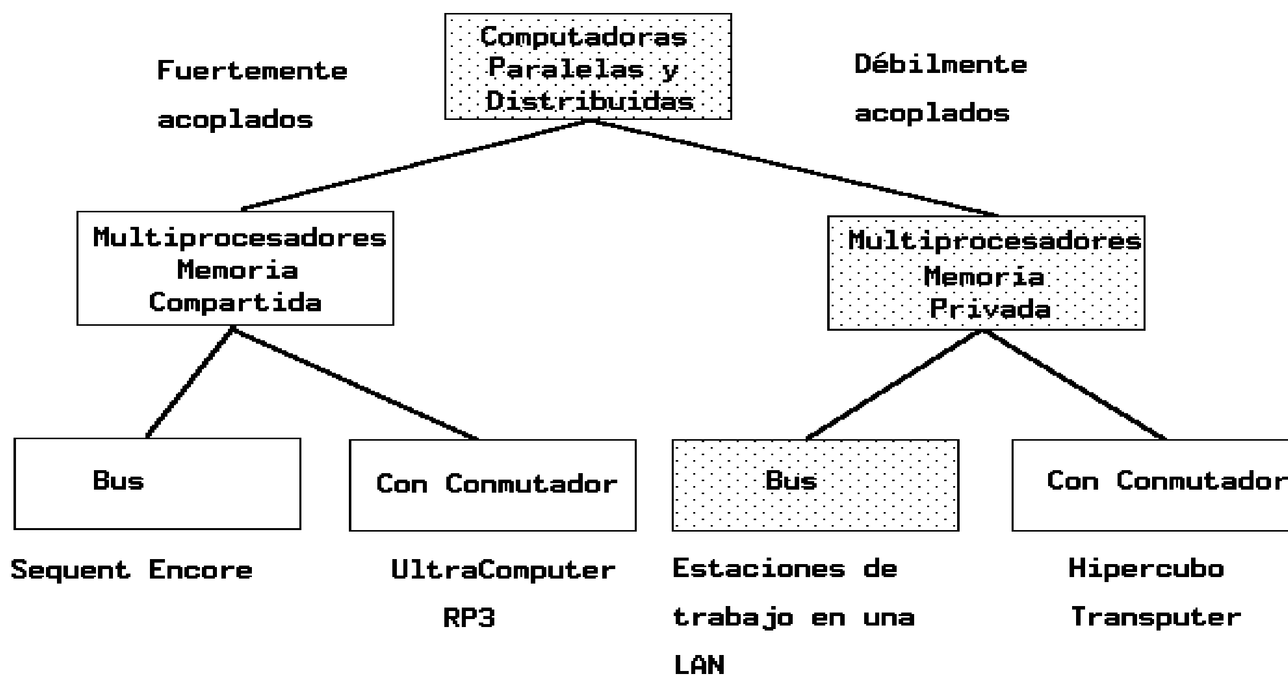


Fig. 20.6. - Clasificación de Computadoras.

20.12.1. - Multiprocesadores con base en buses

Estos sistemas constan de un cierto número de CPUs conectados a un bus común, junto con un módulo de memoria. Esta memoria es *Coherente* ya que si una CPU escribe algo sobre una dirección de memoria y luego otra CPU la lee, obtiene el valor actualizado.

El problema de este esquema es que con 4 o 5 CPU's el bus estará por lo general sobrecargado y el rendimiento disminuirá, para solucionar esto se prevén memorias cachés entre el CPU y el bus. El tamaño tipo de un caché varía desde 64 K hasta 1 M dando como resultado una tasa de hallazgos del 90 % o más.

Para solucionar el problema de la coherencia del caché cuando dos CPU's acceden al mismo dato y lo copian en sus respectivos cachés para luego actualizarlo se ideó que cuando una palabra es escrita en el Cache, también sea escrita en la memoria. Se denomina a este caché, *caché de escritura* (el tráfico en el bus solo es para escritura).

Además todos estos caches monitorean el bus constantemente. Cada vez que un caché observa una dirección de memoria de la cual él tiene una copia puede actualizarla en base al dato que está circulando en el bus. Se denomina *Cachés Monitores* a este tipo de cachés ya que observan al bus y se actualizan automáticamente.

La mayoría de los multiprocesadores basados en buses utilizan esta tipo de arquitectura con cachés de escritura y cachés monitores u otra muy similar, de esta forma es posible colocar desde 32 hasta 64 CPU's en el mismo bus.

20.12.2. - Multiprocesador con conmutador

Para poder conectar más de 64 procesadores es necesario un método distinto de conectar cada CPU a la memoria. Una posibilidad es dividir la memoria en módulos y conectarlos a las CPU con un conmutador Crossbar Switch (ver capitulo 7).

La ventaja de este esquema es que muchas CPU pueden acceder simultáneamente a la memoria, siempre y cuando no accedan al mismo módulo en cuyo caso deberán esperar.

La desventaja es que teniendo n CPU y n Memorias, se necesitan n^2 conmutadores. Si n es grande el costo puede ser prohibitivo.

Otra red posible es la red omega que necesita N etapas de conmutación en la cual con n CPU's y n memorias se necesitan $\log_2 n$ etapas de conmutación cada una de las cuales tiene n/2 conmutadores para un total de (n $\log_2 n$) / 2 conmutadores.

Igualmente si la cantidad de conmutadores es muy alta, por ejemplo n = 1024 con 10 etapas de conmutación la velocidad requerida de cada conmutador para responder a un requerimiento puede obligar a que cada conmutador posea un tiempo de respuesta tal que elevaría el costo individual alcanzando un costo total de la red también prohibitivo.

Otra implementación que intenta reducir un poco los costos es mediante los sistemas jerárquicos en los cuales cada CPU tiene asociada una memoria con rápido acceso y puede acceder pero más lentamente a la memoria de los demás (NUMA). La desventaja de este esquema es que la colocación de los programas y de los datos en el esquema NUMA se convierte en un factor crítico cuando se intenta lograr que la mayoría de los accesos sean a la memoria local.

Como conclusión final se puede decir que la construcción de un multiprocesador grande fuertemente acoplado y con memoria compartida es difícil y cara.

20.12.3. - Multicomputadoras con base en buses (Sistema Distribuido)

Este sistema es de fácil construcción pero tiene el problema de la forma en que se comunican las CPU. Es necesario cierto esquema de interconexión pero como solo es entre CPUs el volumen de tráfico será varios ordenes menor que si se utiliza la red de interconexión para el tráfico CPU - Memoria (por ejemplo de entre 10/100 Mb/seg en el primer caso y 300 Mb/seg en el otro).

Fig. 20.7. - Estaciones de trabajo en una LAN.

20.12.4. - Multicomputadoras con conmutador

Se han propuesto y construido varias redes de interconexión pero todas tienen la propiedad de que cada CPU tiene acceso directo y exclusivo a su propia memoria particular. Ejemplo Retícula, también llamada Red con Vecinos Cercanos o Mesh-Connected, Hipercubo, etc. Estas arquitecturas se adecuan a problemas de naturaleza bidimensional. Actualmente existen hipercubos disponibles comercialmente en el mercado de hasta 16384 CPU's.

20.13. - **CONCEPTOS DE SOFTWARE**

Daremos una introducción a los distintos tipos de sistemas operativos para los multiprocesadores y multicomputadoras. Los sistemas operativos no se pueden clasificar tan fácil como el hardware. A pesar de que el software es algo vago se pueden distinguir dos tipos de sistemas operativos: los débilmente acoplados y los fuertemente acoplados. Esto es un tanto análogo al hardware débilmente y fuertemente acoplado.

En las siguientes secciones analizaremos algunas de las combinaciones más comunes de software y hardware.

20.13.1. - **Sistemas operativos de redes** (Software débilmente acoplado en Hardware débilmente acoplado)

Es una situación en la que cada máquina tiene un alto grado de autonomía y existen pocos requisitos a lo largo de todo el sistema, las personas se refieren a ellas como un *sistema operativo de red*.

Una función importante del sistema operativo de red es permitir a los usuarios hacer un login en forma remota con otra computadora, por ejemplo Internet nos brinda el Telnet para este propósito. Una vez que se ha

establecido la comunicación, el software de la red crea una unión transparente y bidireccional tal que todos los caracteres ingresados por el usuario son mandados al proceso que se está ejecutando en forma remota y todos los datos que salen del proceso son enviados de vuelta al usuario. Esto nos permite realizar operaciones sobre la máquina remota como si fuésemos cualquier usuario local.

Otra función que permiten los sistemas operativos de red es proveer un mecanismo de transferencia de archivos. Esto se implementa por ejemplo contando con una máquina que actúa como **servidor** de archivos y que provee a las otras máquinas de los usuarios **clientes** de servicios para leer o escribir estos archivos.

En los servidores de archivos la información se estructura con alguna jerarquía y los clientes pueden importar o montar esta jerarquía aumentando la ya existente en sus propias máquinas.
Internet brinda este mecanismo de transferencia de archivos por medio del FTP (File Transfer Protocol).

1) FTP remote machine
2) Posicionarse en el directorio donde está el archivo y GET arch_origen arch_destino

El mecanismo FTP es implementado de una manera similar al telnet. Hay un daemon en la máquina remota que observa peticiones de conexión al port del sistema FTP. Comandos que brinda:

GET	Copia un archivo de la máquina remota a la local.
PUT	Copia un archivo de la máquina local a la remota.
LS // DIR	Lista archivos en el directorio actual de la remota.
CD	Cambia de subdirectorio en el nodo remoto.

El sistema operativo a usar en este ambiente debe controlar las estaciones de trabajo en lo individual, a los servidores de archivos y también encargarse de la comunicación entre ellos.

20.13.2. - **Sistema multiprocesador de tiempo compartido** (Software fuertemente acoplado en Hardware fuertemente acoplado)

La característica clave de este tipo de sistema es la existencia de una sola cola para la ejecución, es decir, una lista de todos los procesos en el sistema que no están bloqueados en forma lógica y listos para su ejecución.

Fig. 20.8. - Un multiprocesador con una cola de ejecución.

La cola de ejecución es una estructura de datos contenida en la memoria compartida.

Existen varias máquinas específicas en esta categoría como ser las dedicadas a bases de datos. Operan como un sistema de tiempo compartido pero con varias CPU's.

Un área en donde este tipo de multiprocesador difiere de una red o un sistema distribuido es en la organización del sistema de archivos. Usualmente el sistema operativo cuenta con un sistema de archivos tradicional.

En muchos de estos sistemas se dedica un CPU a la ejecución del sistema operativo y los otros atienden las tareas de los usuarios. Nótese que en estos casos el CPU que ejecuta el S.O. se convierte a menudo en un cuello de botella.

20.13.3. - **Sistema realmente distribuido** (Software fuertemente acoplado en Hardware débilmente acoplado).

El objetivo es crear la ilusión en la mente de los usuarios de que toda la red es un solo sistema de tiempo compartido.

Definición de Sistema Distribuido: Es aquel que se ejecuta en una colección de máquinas sin memoria compartida pero que aparece ante sus usuarios como una sola computadora (uniprocesador virtual).

Características:

- Debe existir un Mecanismo de comunicación global entre los procesos (cualquiera puede hablar con cualquiera). No tiene que haber distintos mecanismos en distintas máquinas o distintos mecanismos para la comunicación local o la comunicación remota.
- Debe existir un esquema global de protección

- La administración de procesos debe ser la misma en todas partes (crear, destruir, iniciar, detener)
- Debe existir un sistema global de archivos y debe tener la misma apariencia en todas partes

Elemento	**S.O. De Red**	**S.O. Distribuido**	**S.O. Multiprocesador**
Se ve como un uniprocesador virtual ?	No	Si	Si
Mismo Sistema Operativo en todas ?	No	Si	Si
Cuantas copias del S.O. existen ?	N	N	1
Cómo se logra la Comunicación ?	Archivos compartidos	Pasaje de mensajes	Memoria compartida
Requiere acuerdos de Protocolos de la red?	Si	Si	No
Existe una única cola de ejecución ?	No	Si	Si
Existe una semántica bien definida para archivos compartidos ?	Generalmente No	Si	Si

20.14. - **SISTEMAS DISTRIBUIDOS**

Definiremos a los Sistemas Distribuidos como sistemas "débilmente acoplados", donde los procesadores no comparten memoria común ni reloj y donde cada procesador tiene su propia memoria local.

La comunicación entre ellos se realiza por medio de vías de comunicación (buses, líneas telefónicas, etc.).

Los objetivos de un Sistema Distribuido son:
- compartir recursos,
- mayor capacidad de procesamiento, y
- comunicación.

La mayor capacidad de procesamiento se da cuando es posible dividir una tarea en varias subtareas concurrentes. Si es posible que éstas ejecuten en distintos computadores se obtiene un alto grado de paralelismo.

Otra situación se da cuando una computadora que se encuentra sobrecargada de trabajo lo deriva a otros procesadores.

Compartir recursos significa la posibilidad de utilización de un recurso no disponible en forma local, pero sí hacer posible su uso en forma remota. Además sería posible la utilización de recursos, descompuestos en un sitio, por otros de igual capacidad en forma remota.

La Comunicación significa el intercambio de información, y abarca desde distribuir procesos hasta el simple correo electrónico.

20.14.1 - **MODOS DE PROCESAMIENTO**.

Los Sistemas Distribuidos permiten La Migración de Datos, La Migración de Procesos y La Migración de Trabajos.

- **Migración de Datos** : cuando un usuario remoto requiere un archivo existen dos maneras de satisfacerlo. Una es enviarle el archivo entero y la otra es enviarle la porción que necesita, si luego necesita más se le enviará. En el primer caso cuando haya modificado el archivo lo reenviará completo, en el segundo reenviará las porciones modificadas.
- **Migración de Procesos** : en muchos casos es preferible, cuando los archivos son muchos o de gran volumen, transferir los procedimientos. Aquí también tenemos dos alternativas : una es que un procedimiento invoque una necesidad a otro en el sitio remoto y que éste la satisfaga (Remote Procedure Call) y la otra es que un proceso envíe un mensaje al sitio remoto y allí se genere un nuevo proceso que satisfaga sus necesidades.
- **Migración de Trabajos** : significa la transferencia directa de Trabajos de un nodo a otro, con lo cual se puede lograr :
  - balancear el sistema
  - subdivisión de tareas que ejecutan en forma concurrente
  - uso de mejor o nuevo hardware
  - uso de mejor o nuevo software

20.15. - **ASPECTO DEL DISEÑO**

20.15.1. - Transparencia

Se trata de lograr la imagen de un único sistema, que todas las personas piensen que la colección de máquinas sea un sistema de tiempo compartido de un solo procesador. Se puede lograr en dos niveles distintos:

1) Ocultar la distribución a los usuarios.
El usuario no tiene por qué saber que la ejecución, por ejemplo, de una serie de compilaciones se realiza en paralelo en distintas máquinas.
2) Transparencia al sistema para los programas.
Lograr diseñar la interfaz de llamadas al sistema de modo que no sea visible la existencia de varios procesadores.
El concepto de transparencia se puede aplicar a distintos aspectos de un sistema distribuido :

Transparencia de localización	Los usuarios no pueden indicar la localización de los recursos
Transparencia de migración	Los recursos se pueden mover de una localización a otra sin cambiar sus nombres
Transparencia de réplica	Los usuarios no pueden indicar el número de copias existentes. El S.O. puede hacer copias de los archivos sin que los usuarios lo noten.
Transparencia de concurrencia	Varios usuarios pueden compartir recursos de manera automática. El S.O. asegura un acceso secuencial no concurrente.
Transparencia de paralelismo	Las actividades pueden ocurrir paralelamente sin que los usuarios lo sepan.

20.15.2. - Flexibilidad

Es importante que el sistema sea flexible ya que las decisiones de diseño que parecen hoy razonables podrían demostrar luego ser incorrectas.

Existen dos escuelas de pensamiento en las estructuras de un SO:

- una mantiene que cada máquina debe ejecutar un núcleo tradicional que proporcione la mayoría de los servicios (*núcleo monolítico*);
- la otra sostiene que el núcleo debe proporcionar lo menos posible y que el grueso de los servicios del sistema operativo se obtenga a partir de los servidores a nivel usuario (*micronúcleo*).

El primero es el SO centralizado básico actual aumentado con capacidades de red y la integración de servicios remotos. La mayoría de las llamadas al sistema se realizan mediante interrupciones al núcleo en donde se realiza el trabajo para después devolver el resultado deseado al proceso del usuario. La mayoría de las máquinas tienen disco y administran sus propios sistemas locales de archivos (extensiones e imitaciones de Unix).

El segundo (microKernel) es el más flexible pues casi no hace nada. Proporciona cuatro servicios mínimos

1) Un mecanismo de comunicación entre procesos.
2) Cierta administración de la memoria.
3) Una cantidad limitada de planificación y administración de procesos de bajo nivel.
4) Entrada/Salida de bajo nivel.

No provee (a diferencia del monolítico) el sistema de archivos, el sistema de directorios, la administración completa de los procesos ni gran parte del manejo de las llamadas al sistema.

Todos los servicios del SO se implementan por lo general como servidores a nivel usuario. Es precisamente esta capacidad de añadir, eliminar y modificar servicios lo que da al microKernel su flexibilidad.

La única ventaja del núcleo monolítico es el rendimiento. Las interrupciones al núcleo y la realización de todo el trabajo allí puede ser más rápido que el envío de mensajes a los servidores remotos.

20.15.3. - Confiabilidad

La idea es que si una máquina falla alguna otra se encargue del trabajo. La confiabilidad global del sistema es mayor que la confiabilidad de un servidor individual. Uno de los aspectos de la confiabilidad es la disponibilidad que se refiere a la fracción de tiempo que se puede utilizar el sistema. Otra herramienta para el mejoramiento de la disponibilidad es la redundancia (duplicar piezas de hard y soft). Un sistema altamente confiable debe ser altamente disponible, y además los datos confiados al sistema no deben perderse o mezclarse de manera alguna (todas las copias deben ser consistentes).

Otro aspecto de la confiabilidad general es la seguridad. Los archivos y otros recursos deben ser protegidos contra el uso no autorizado. Esto es similar a los métodos vistos pero es más severo en los sistemas distribuidos. No es sencillo determinar de quién proviene una solicitud "mensaje", lo que requiere un cuidado considerable.

Otro aspecto es la tolerancia de fallas. Si un servidor falla y vuelve a arrancar de manera súbita, si el servidor tenía tablas con información importante respecto de las tareas en curso lo menos que puede ocurrir es que la recuperación será difícil .

20.15.4. - Desempeño o Performance

En particular cuando se ejecuta una aplicación en un sistema distribuido no debe parecer peor que su ejecución en un único procesador.

Se pueden utilizar diversas métricas del desempeño, por ejemplo :

- Tiempo de respuesta.

- Rendimiento (cantidad de trabajos por hora).
- Uso del sistema.
- Cantidad consumida de la capacidad de la red.

El problema del desempeño se complica por el hecho de que la comunicación (factor esencial en un sistema distribuido) es algo lenta por lo general.

La mayoría de este tiempo se debe a un manejo inevitable que realizan los protocolos en ambos extremos en lugar del tiempo que tardan los bits en viajar por el medio de transmisión.

La tolerancia a fallas también afecta el desempeño.

20.15.5. - Escalabilidad

La mayoría de los sistemas distribuidos están capacitados para trabajar con unos cuantos cientos de CPU's. No sería una buena idea tener un solo servidor de correo para cincuenta millones de usuarios, aunque tuviera la suficiente capacidad de CPU y almacenamiento para ello, la capacidad de la red dentro y fuera de él sí sería un problema. Además no toleraría bien los fallos. Las tablas centralizadas son tan malas como los componentes centralizados. Los algoritmos centralizados también son una mala idea, el problema es la recolección de la información para el algoritmo.

En los algoritmos descentralizados:

1) Ninguna máquina tiene la información completa acerca del estado del sistema.
2) Las máquinas toman decisiones solo en base a la información disponible de manera local.
3) El fallo de una máquina no arruina el algoritmo.
4) No existe una hipótesis implícita de la existencia de un reloj global. (ver capítulo de Sincronización)

20.15.6. - **Ventajas y Desventajas**

Puntos a favor:

- buena proporción precio/desempeño y se pueden ajustar bien a las aplicaciones distribuidas,
- pueden ser altamente confiables y aumentar su tamaño en forma gradual al aumentar la carga de trabajo.

Puntos en contra:

- software más complejo
- potenciales cuellos de botella en la comunicación
- la seguridad es débil

Los modernos sistemas de cómputos tienen con frecuencia varios CPU. Se pueden organizar con multiprocesadores (memoria compartida) o multicomputadoras (sin memoria compartida), ambos se pueden basar en un bus o en un conmutador, los primeros suelen ser fuertemente acoplados mientras que los segundos débilmente acoplados.

Los sistemas distribuidos deben diseñarse con cuidado, teniendo en cuenta la transparencia, la flexibilidad, confiabilidad, desempeño y escalabilidad. En los sistemas operativos de red, los usuarios conocen la existencia de múltiples máquinas y es necesario para acceder a estos recursos un login en la máquina remota apropiada, o transferir datos de la remota a su propia máquina. Mientras que en los sistemas operativos distribuidos, los usuarios no necesitan tener en cuenta las múltiples máquinas; ellos acceden a prestaciones remotas de la misma forma que a las locales.

El software se divide en tres clases:

- Sistemas operativos de red (estación de trabajo independiente)
- Sistemas operativos distribuidos (convierten toda la colección de hardware y software en un único sistema integrado).
- Multiprocesadores con memoria compartida (no son sistemas distribuidos).

20.16. - **COMUNICACIONES**

La mayor y más importante diferencia entre un sistema distribuido y un uniprocesador es el mecanismo de comunicación entre procesos. Para que la comunicación sea coherente, existen protocolos.

20.17. - **ESTRATEGIAS DE DISEÑO**

Niveles de un Protocolo

Cuando A se quiere comunicar con B, primero arma el mensaje y luego ejecuta un System Call y el S.O. toma el mensaje y lo envía a través de la red. Para esto A y B se tienen que poner de acuerdo si los bits se toman de izquierda a derecha, si se mandan en ASCII o EBCDIC, etc.

Para facilitar el trabajo con los distintos niveles y aspectos correspondientes a la comunicación, la organización internacional de estándares (International Standards Organization -ISO-) ha desarrollado un modelo de refe-

rencia que identifica en forma clara los distintos niveles, les da nombres estandarizados y señala cuál nivel debe realizar cada trabajo. Este modelo se llama el *modelo de referencia para interconexión de sistemas abiertos* (Day y Zimmerman, 1983), lo cual se abrevia por lo general como **ISO OSI** o a veces solo como **modelo OSI**.

El modelo OSI está diseñado para permitir la comunicación de los sistemas abiertos. Se dice que un sistema es abierto cuando está preparado para comunicarse con cualquier otro sistema abierto mediante estándares que gobiernan el formato, contenido y significado de los mensajes enviados y recibidos. Básicamente un protocolo es un acuerdo en cómo debe llevarse la comunicación. El modelo OSI distingue entre dos tipos de protocolos:

Orientados a conexión: Establecen una conexión explícita entre Sender y Receiver y posiblemente negociación de qué protocolo van a utilizar (Ej. Teléfono)

Orientados sin conexión: No se realiza ningún tipo de negociación previa, se envían los datos cuando el Sender está listo.

El modelo OSI se divide en siete capas/niveles. Cada capa se encarga de un aspecto específico de la comunicación. De esta forma, el problema se puede dividir en piezas manejables, cada una de las cuales se puede resolver en forma independiente de las demás. Cada capa proporciona una *interfase* con la otra capa por encima de ella. La interfase es un conjunto de operaciones que juntas definen el servicio que la capa está preparada para ofrecer a sus usuarios. Estas capas son:

1) *Física* : Responsable de los detalles físicos
2) *Data Link* : Responsable de la comunicación de la red (protocolos)
3) *Network* : Responsable de los paquetes (asegura la trayectoria)
4) *Transport* : Responsable del transporte de paquetes con un orden
5) *Sesión* : Sirve como interfase entre el usuario y el servicio de transporte.
6) *Presentación* : Homogeneización de datos y de dispositivos (criptografía, compresión, etc.)
7) *Aplicación* : Responsable del manejo de datos. Concierne al soporte de aplicación del usuario.

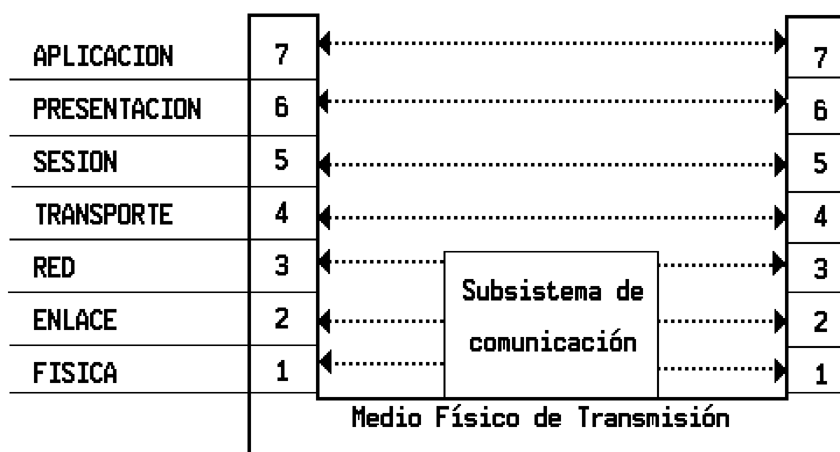


Fig. 20.9. - El modelo en capas OSI/ISO.

20.17.1. - Hardware Layer o Capa Física

Es la que se encarga de transmitir los ceros y unos a la máquina vecina, cuántos bits por segundo transmite, el tipo de conector de red, si la transmisión es bidireccional simultáneamente o no, etc. El protocolo de la capa física se encarga de la estandarización de las interfases eléctricas mecánicas y de señalización. Por ejemplo uno de los estándares desarrollados para esta capa es la comunicación serial RS-232.

20.17.2. - Data Link Layer o Capa de Enlace

Esta capa detecta errores de transmisión es decir agrupa un conjunto de bits en frames (tramas, marcos) y chequea que cada frame sea recibido correctamente (pone información extra en el mensaje y agrega un checksum (suma de verificación), además todas las tramas tienen un número de secuencia como etiqueta.

20.17.3. - Network Layer o Capa de Red

Para que un mensaje llegue del emisor al receptor, tienen que hacer un cierto numero de saltos y en cada uno de ellos elegir un nuevo nodo al cual ir (hops). Cómo elegir un camino se llama Routing (ruteo) y es el principal trabajo de esta capa (asegura la trayectoria). Para esto hay que tener en cuenta el tráfico y cuántos mensajes existen encolados para salir por una línea en cada nodo.

Un protocolo conocido orientado a conexión de esta capa es el **X.25** que es el utilizado en grandes redes (WANs). Al inicio el usuario de X.25 envía una solicitud de llamada a un destino, el cual puede aceptar o rechazar la conexión propuesta. Si se acepta la conexión, quien hace la llamada obtiene un identificador de conexión para usarlo en las solicitudes posteriores. En muchos casos la red escoge una ruta al receptor durante esta configuración y la utiliza para el tráfico posterior.

Otro protocolo pero sin conexión es el **IP** (Protocol Internet) y es parte de los protocolos DoD (Departamento de Defensa de los Estados Unidos). Un paquete IP se envía sin configuración alguna.

Cada paquete tiene una ruta hacia su destino independiente de los otros paquetes, e inclusive pueden tomar distintos caminos para llegar a destino. No se selecciona ruta alguna ni se recuerda ésta, como ocurre a menudo en X.25.

Estas tres capas anteriores se encuentran involucradas indefectiblemente en todos los nodos del camino que toma el mensaje, las cuatro siguientes solo en el emisor y receptor (origen y destino).

20.17.4. - Transport Layer o Capa de Transporte

El trabajo de esta capa es proveer confiabilidad a nivel de proceso, es decir estar seguro que tarde o temprano va a llegar a destino, para esto el mensaje que recibe de la capa sesión lo fracciona, le asigna un número secuencial y después envía todos. Se puede decir que es responsable del transporte de paquetes manteniendo su orden (fragmentación de paquetes) para que estos signifiquen un mensaje.

Estos tipos de protocolos pueden ser implementados sobre X.25 o IP. En el primer caso, los paquetes llegarán en orden mientras que en el segundo, la capa de transporte debe reordenarlos antes de pasarlos a la aplicación que va a recibirlos.

El protocolo de transporte oficial ISO tiene cinco variantes conocidas como TP0 a TP4. El protocolo de transporte DoD orientado a conexión se llama **TCP** (Transmission Control Protocol). Es similar a TP4. La combinación TCP/IP es muy utilizada en universidades y en la mayoría de los sistemas UNIX. La serie de protocolos DoD también soportan un protocolo de transporte sin conexión llamado **UDP** (Universal Datagram Protocol) que en esencia es igual a IP con ciertas adiciones menores. Los programas del usuario que no necesitan un protocolo orientado a conexión utilizan por lo general UDP.

20.17.5. - Session Layer o Capa de Sesión

Es una refinación de la capa de transporte, mantiene el registro de quién está hablando y provee facilidades de sincronización. Esto último permite a los usuarios colocar Checkpoints en las transmisiones largas (por si se cae la red). Generalmente esta capa no se implementa.

20.17.6. - Presentation Layer o Capa de Presentación

Esta capa es la encargada del significado de los bits, permite la homogeneización de datos, facilita la comunicación entre las máquinas con distintas representaciones internas, o sea que ambas partes entiendan lo mismo de un mensaje. En esta capa se pueden definir registros que contengan campos como los mencionados y que entonces el emisor notifique al receptor que un mensaje contiene un registro particular en cierto formato. Ej. Criptografía, Compresión y Conversión de datos.

20.17.7. - Application Layer o Capa de Aplicación

Esta capa es la que interactúa con los usuarios, brindando una colección de protocolos misceláneos para actividades comunes como ser File Transfer (FTP), Correo Electrónico, Remote Logins, Telnet's. Los más conocidos de estos protocolos son el X.400 de correo electrónico y el servidor de directorios X.500.

20.18. - **ESTRATEGIAS DE RUTEO**

Se llama ruteo a una sucesión de nodos o terminales o gateways que deben ser recorridos para el envío de mensajes a través de la red.

Las estrategias empleadas pueden ser:

Rutas Fijas : El camino entre dos nodos es fijo o sea que el envío de los mensajes se realiza por el mismo camino.

Circuito Virtuales : El camino entre dos nodos es fijo durante una sesión de mensajes y puede variar en otras. O sea que cuando se toma un camino para enviar no se lo libera hasta su fin, pero si después es necesario volver a mandar mensajes al mismo nodo, se puede tomar otro camino distinto, logrando con esto tal vez evitar un embotellamiento de paquetes.

Ruteo Dinámico : El camino entre dos nodos se determina en el momento de enviarse el mensaje (cada mensaje). Esto implica que los paquetes consecutivos viajan por distintos caminos.

20.19. - **ESTRATEGIA DE COMUNICACIÓN**

Estas estrategias indican cómo son conectados dos nodos a través de la red. Pueden ser:

Conmutación de circuitos : Se establece una conexión física fija en cada sesión de comunicación, de forma similar a la que se utiliza en el sistema telefónico.

Conmutación de mensajes : Se establece una conexión fija durante toda la transmisión de un mensaje.

Conmutación de Paquetes : Los mensajes son "cortados" en paquetes y cada paquete puede seguir caminos distintos por la red.

20.19.1. - **Disciplina de Prioridades y Protocolos**

En cuanto al acceso al medio (línea de transmisión) siempre es posible definir una Disciplina de Prioridades que establezca el orden en el cual cada nodo accederá a la línea de transmisión.

Los diferentes tipos de Disciplinas de prioridades se basan en métodos del estilo del Round-Robin o Prioridades (que pueden ser estáticas o dinámicas) explicados ya en capítulos anteriores

Asimismo definimos como Protocolo al conjunto de reglas de control que garanticen al nodo de mayor prioridad su acceso al medio independientemente de la Disciplina de Prioridades que se utilice. En consecuencia un mismo Protocolo puede servir a diferentes Disciplinas de Prioridades.

20.19.1.1.- **Ranuras**

El tiempo de uso del medio de transmisión puede ser dividido en intervalos denominados Ranuras o Slots de una cierta duración fija o variable.

La Ranura de Tiempo (Slot Time) es el período de tiempo límite durante el cual podría ocurrir una colisión. Para determinar el tiempo de ranura es muy importante el tiempo de Propagación del mensaje en la línea de conexión.

La disciplina de prioridades y el protocolo de acceso deben hacer posible asignar siempre una ranura a un determinado nodo a efectos de aprovechar al máximo el medio de transmisión.

20.20. - **CLASIFICACION DE PROTOCOLOS**

Los protocolos pueden clasificarse según la forma en la cual administran el acceso al medio de transmisión como Protocolos de acceso Controlado y Protocolos de acceso Contencioso.

20.20.1. - **Protocolos de Acceso Controlado**

20.20.1.1. - **Protocolos con Mecanismo de Reserva**

Uno de los tipos de protocolo de acceso controlado es el que implementa Mecanismo de Reserva. En tales protocolos la ranura se compone de dos partes, una de las cuales está destinada a contener las reservas que van realizando los nodos de las ranuras futuras y la otra parte de la ranura contiene el mensaje.

La ranura de reserva está compuesta por miniranuras cada una de las cuales corresponde a un nodo de la red. Cuando un nodo desea hacer uso del medio expresa su intención de hacerlo indicándolo en la miniranura que le está asignada. El tiempo de la miniranura se establece como el tiempo total de propagación de la reserva desde un extremo a otro de la red.

En consecuencia cuando termina el período de reserva todos los nodos conocen cuáles son los que desean transmitir y por lo tanto ceden el derecho de acceso para la transmisión al nodo de mayor prioridad según la Disciplina de Prioridades establecida.

20.20.1.2. - **Protocolo de Paso de Token (Token Passing)**

En esta clase de protocolos el acceso al medio se controla por medio del pasaje de un nodo a otro de una ficha o token.

El pasaje de una ficha real implica necesariamente una topología de anillo el cual puede ser real (anillo, token ring) o virtual.

El mensaje que se transmite por la línea se denomina trama. En el protocolo con token en la trama existe un encabezamiento que indica si lo que sigue a continuación es un mensaje válido o si la trama está vacía y por ende disponible para que un nodo transmita un mensaje.

El nodo que detecta que la trama está vacía y desea transmitir un mensaje coloca el encabezamiento de mensaje válido, le agrega el mensaje en cuestión y transmite la trama en el medio. Cuando el mensaje ha dado la vuelta completa al anillo el nodo que originalmente lo transmitió lo retira del anillo y devuelve al mismo la trama con encabezamiento de trama vacía cediendo así el token al siguiente nodo en el anillo.

En el caso del anillo virtual o lógico los nodos están conectados a una barra o bus bidireccional (la denominada topología de Bus compartido o Shared Bus). El anillo queda definido porque cada nodo conoce a su predecesor y a su antecesor. El protocolo de pasaje de token para estas topologías se denomina Token Bus.

El protocolo para la topología Token Bus fue reglamentado por la IEEE en su norma 802.4, en tanto que el protocolo para Token Ring corresponde a la norma IEEE 802.5.

20.20.1.3. - **Protocolo de Paso de Token con Slots**

Una forma de que circulen varios mensajes en el anillo consiste en dividir el anillo en varias ranuras. El mecanismo de funcionamiento es en todo similar al del Paso de Token anterior al cual se le debe agregar, por el hecho de que varios mensajes están circulando simultáneamente por el anillo, el hecho de que cada nodo debe ir llevando la cuenta de la cantidad de mensajes que han pasado desde que él envió el suyo a efectos de poder detectar la vuelta del mensaje original que despachó (debe conocer la cantidad de mensajes que circulan en el anillo).

20.20.2. - **Protocolos de Acceso Contencioso**

En estos protocolos un nodo sensa la línea de transmisión y cuando percibe silencio en la misma transmite su mensaje. Puede suceder que dos o más nodos detecten silencio en la línea y transmitan ambos sus propios mensajes, en ese caso se produce lo que se denomina una *colisión*. Tal colisión es escuchada por los nodos que han transmitido los mensajes, los cuales abortan sus respectivos intentos para reintentarlo nuevamente luego de un tiempo generado aleatoriamente. En la Fig. 20.10 puede verse la lógica de funcionamiento de tal mecanismo.

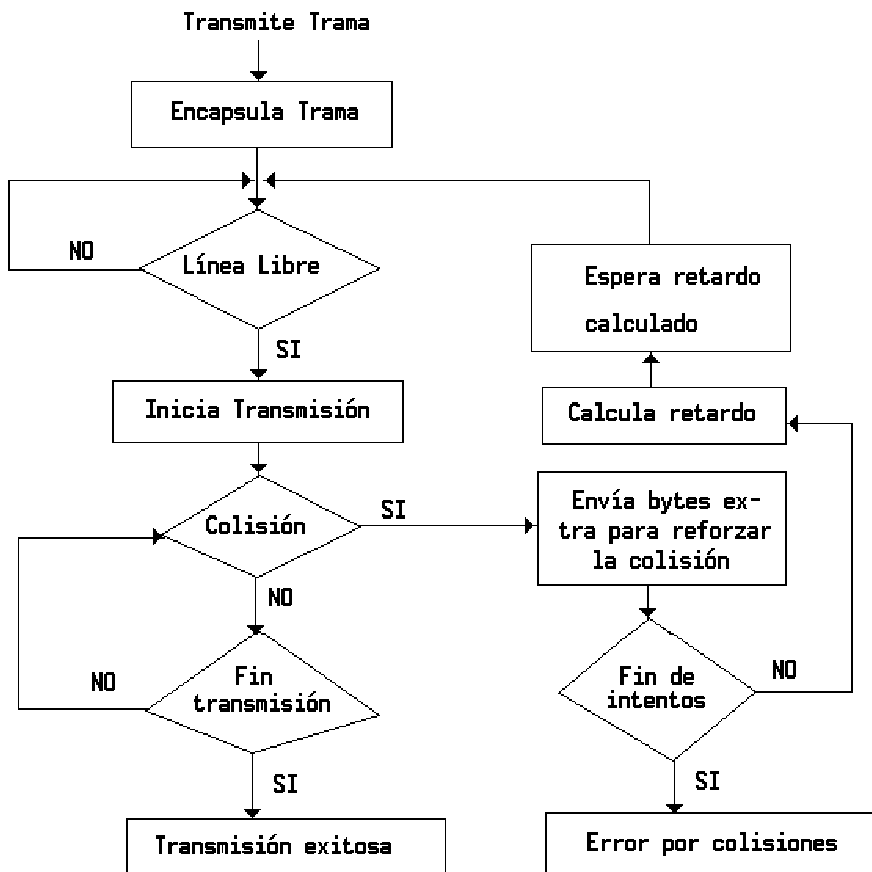


Fig. 20.10. - Comunicación en Protocolos Contenciosos.

20.20.2.1. - **Protocolo CSMA/CD**

En los protocolos de Acceso Múltiple con Sensado de Portadora (del inglés Carrier Sense Multiple Access, CSMA, con Detección de Colisiones -with Collision Detection-) las estaciones escuchan o sensan el canal de transmisión y usan esa información para determinar la posibilidad o no de transmitir.

Existen dos variantes: no-persistente y p-persistente.

20.20.2.1.1. - **CSMA no-persistente**

En este protocolo el nodo escucha la línea si no está ocupada transmite y si está ocupada posterga la transmisión de acuerdo a una cierta política preestablecida de demoras. Estos protocolos pueden ser ranurados o no.

En los protocolos ranurados un nodo puede transmitir solo al comienzo de una ranura. Un mensaje generado en el transcurso de una ranura solo puede transmitirse al inicio de la siguiente. En este caso los mensajes no colisionan o colisionan totalmente.

20.20.2.1.2. - **CSMA p-persistente**

En este protocolo la estación sensa el canal. Si está ocupado demora la transmisión como en el no-persistente. Pero si la línea está libre transmite con probabilidad p. Con probabilidad 1-p espera un tiempo fijo (en general el tiempo de propagación de la señal entre los extremos de la red) al final de lo cual repite el procedimiento anterior. Cuanto menor es p menor es la probabilidad de colisiones.
Ethernet que es un protocolo contencioso es un CSMA 1-persistente.

20.20.2.1.3. - **CSMA/CA**

Otra variante del protocolo CSMA es el CSMA/CA (with Collision Avoidance) en el cual luego de una transmisión se reservan ranuras específicas para cada uno de los nodos en las cuales pueden transmitir sin colisión.

20.21. - **TIPOS DE SISTEMAS**.

Para clasificar los tipos de sistemas existentes un enfoque de acuerdo a la repartición geográfica parece lo más adecuado.

Redes Globales (GAN - Global Area Network): designan generalmente las llamadas redes geográficas, o sea redes que pueden abarcar un país o varios.

Existen dos grandes tipos :

- Las redes de **Teleproceso** que históricamente son las más antiguas, basadas en un computador central al cual se conectan terminales remotas vía módem en topologías estrella o bus compartido.
- Las redes generales de **Conmutación de Paquetes** : basadas en la técnica de conmutación de paquetes que se multiplexan y pueden viajar por diferentes caminos hacia el nodo destino.

Redes Locales (LAN - Local Area Network) : que designan redes de computadores que no están a demasiada distancia entre sí, generalmente con aplicaciones en la oficina e industria.

Redes de Campo (FAN - Field Area Network o SAN - Small Area Network) : realizan la conexión de microcontroladores que operan directamente sobre dispositivos asociados a robots, autómatas, procesos industriales, etc.

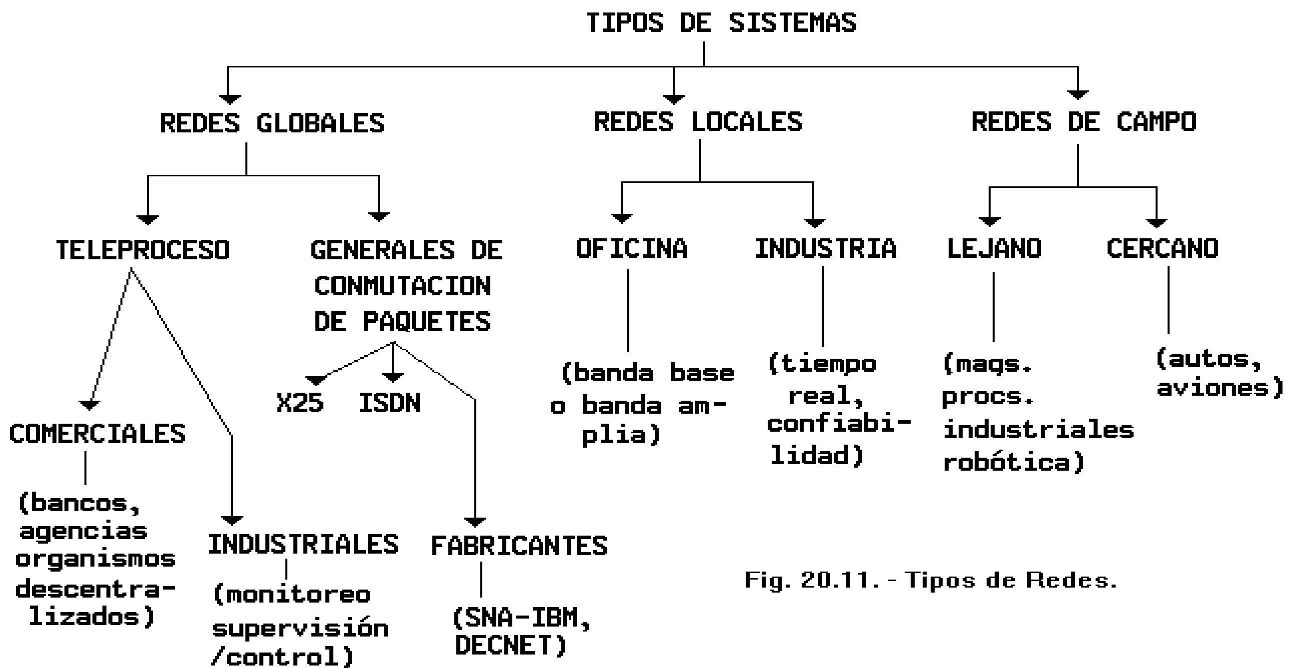


Fig. 20.11. - Tipos de Redes.

CAPITULO 21

MODELO CLIENTE SERVIDOR

21.1. **Introducción**

Como vimos en el modelo ISO son necesarias siete capas con el consecuente encabezado de cada una de ellas en el mensaje, cosa que analizar y construir estos encabezados lleva tiempo. Si bien esto en las redes de área amplia no es significante, si lo es en una LAN.

Para el caso CLIENTE-SERVIDOR, se utiliza un protocolo solicitud-respuesta (request/reply), en vez del OSI (TCP/IP). El cliente envía un request pidiendo un servicio y el server lo recibe, realiza el trabajo y devuelve los datos pedidos o un código de error. Lo principal es su sencillez y su eficiencia.

Sencillez : No se tiene que establecer ninguna conexión sino hasta que esta se utilice, y el mensaje de respuesta sirve como agradecimiento a la solicitud.

Eficiencia : Las capas del protocolo son menos y por lo tanto más eficiente, si todas las máquinas fuesen idénticas, solo se necesitarían tres niveles: La Física, La de Enlace (ambas manejadas por Hardware), la de Solicitud/Respuesta (en lugar de la de sesión).

Las capas 3 y 4 no se utilizan pues no es necesario el ruteo ni tampoco se establecen conexiones. No existe administración de la sesión puesto que no existe y tampoco se utilizan las capas superiores.

Debido a que esta estructura es sencilla, se pueden reducir los servicios de comunicación que presta el micronúcleo a dos llamadas a sistema:

- SEND (dest, &mptr): envía el mensaje que apunta mptr al proceso destino y bloquea al proceso llamador hasta que se envíe el mensaje;
- RECEIVE (addr, &mptr): que hace que se bloquee el que realizó la llamada, hasta que llegue un mensaje, cuando llega este se copia en el buffer que apunta mptr y quien hizo la llamada se desbloquea. El parámetro addr, determina la dirección a la cual escucha el receptor.

21.2. - **Direccionamiento**

Hay varios métodos para el direccionamiento de los procesos:

1) *Integrar machine.number al código del proceso cliente* : machine indica el número de máquina dentro de la red y number, el número de proceso dentro de esa máquina. Pero este método no posee la transparencia que se busca ya que se está identificando que existen varias máquinas trabajando. A parte si se descompone esa máquina (server) se pierde el servicio pues los programas compilados tienen integrado ese número de máquina para ese servicio. Una variación de este esquema, utiliza machine.local_id.

2) *Dejar que los procesos elijan direcciones al azar y localizarlos mediante transmisiones* : En una LAN que soporte transmisiones, el emisor puede transmitir un paquete especial de localización con la dirección del proceso destino, todos los núcleos de las máquinas en la red reciben este mensaje y verifican si la dirección es la suya; en caso de que lo sea, regresa un mensaje "aquí estoy" con su dirección en la red (número de máquina). El núcleo emisor utiliza entonces esa dirección y la captura para uso posterior. Si bien esto cumple con las premisas, genera una carga adicional en el sistema.

3) *Generar un servidor de nombres* : Cada vez que se ejecute un cliente en su primer intento por utilizar un servidor, el cliente envía una solicitud al servidor de nombres (nombre en ASCII) para pedirle el número de la máquina donde se localiza el servidor. Una vez obtenida la dirección se puede enviar la solicitud de manera directa. El problema de este método es que es un componente centralizado y si bien se puede duplicar, presenta problemas en el mantenimiento de la consistencia.

Un método totalmente distinto, utiliza un hardware especial, dejando que los procesos elijan su dirección en forma aleatoria.

Sin embargo en vez de localizarlo mediante transmisiones en toda la red, los chips de interfase de la red se diseñan de modo que permitan a los procesos almacenar direcciones de procesos en ellos. Entonces las tramas usarían direcciones de procesos en vez de direcciones de máquinas. Al recibir cada trama, el chip de interfase de la red solo tendría que examinar la trama para saber si el proceso destino se encuentra en esa máquina. En caso afirmativo, la aceptaría, de lo contrario no.

21.3. - **PRIMITIVAS BLOQUEANTES vs NO BLOQUEANTES**

21.3.1. - **SEND BLOQUEANTES** (Primitivas sincrónicas)

Mientras que se envía el mensaje el proceso emisor se bloquea, de manera análoga el RECEIVE. En algunos casos el receptor puede especificar de quiénes quiere recibir el mensaje y queda bloqueado hasta que reciba el mensaje de él. La CPU está muerta, desperdiciando tiempo.

21.3.2. - **SEND NO BLOQUEANTES** (Primitivas asincrónicas)

Regresa de inmediato el control a quien hizo la llamada (send) antes de enviar el mensaje. La ventaja de esto es que puede trabajar en forma paralela con el envío del mensaje. Sin embargo existe la desventaja de no poder usar el buffer hasta que no se envíe la totalidad del mensaje. Una forma de solucionar esto es que el S.O. copie este buffer a una área propia y luego envíe el mensaje, liberando el buffer. Aquí se desperdicia tiempo en la copia.

21.3.3. - **SEND SIN BLOQUEO CON INTERRUPCIÓN**

El emisor es interrumpido cuando el mensaje fue enviado y el buffer está disponible. No se requiere de una copia, lo que ahorra tiempo, pero las interrupciones a nivel usuario dificultan mucho la programación. Maximiza el paralelismo.

En condiciones normales la primera opción es la mejor, no maximiza el paralelismo pero es fácil de comprender e implantar y no requiere el manejo del buffer en el núcleo. Generalmente el uso de SENDs bloqueantes o no bloqueantes se deja a los diseñadores, pues esto está muy ligado al problema que quieren solucionar. Aunque en algunos casos se dispone de los dos y el usuario elige su favorito.

21.4. - **PRIMITIVAS ALMACENADAS EN BUFFER vs NO ALMACENADAS**

Todas las primitivas anteriormente mencionadas son primitivas no almacenadas. Las primitivas almacenadas en buffer aparecen como una solución a una diferencia entre el Cliente y el Server en el envío y recepción de mensajes, o sea que si en las primitivas anteriormente mencionadas, el cliente envía un SEND antes de que el server ejecute un RECEIVE para esa dirección, se produce una pérdida del mensaje y con resultado de esto, el cliente reintentará la petición.

Si el cliente reintenta muchas veces puede ser que considere que el server se ha caído, o que la dirección no es correcta.

Entonces las primitivas almacenadas en buffer implican que el núcleo del receptor mantenga pendiente los mensajes por un instante usando un buffer lo que conlleva a un almacenamiento y manejo de mensajes que van llegando en forma prematura.

Una forma conceptualmente sencilla de enfrentar este manejo de los buffers es definir una nueva estructura llamada *buzón*.

Un proceso interesado en recibir mensajes le indica al núcleo que cree un buzón para él y especifica una dirección en la cual buscar los paquetes de la red. Así, todos los mensajes que lleguen a esa dirección se colocan en el buzón. El Receive elimina ahora un mensaje del buzón o se bloquea si no hay un mensaje presente.

En ciertos sistemas, no se deja que un proceso envíe un mensaje si no existe espacio para su almacenamiento en el destino. Para que este esquema funcione, el emisor debe bloquearse hasta que obtenga de regreso un reconocimiento, que indica que el mensaje ha sido recibido.

21.5. - **PRIMITIVA CONFIABLES vs NO CONFIABLES**

Confiable : Cuando se garantiza de alguna manera que el mensaje llega a destino.
No Confiable : Cuando no existe garantía alguna de que el mensaje hay sido entregado, podría haberse perdido.

Existen tres distintos enfoques de este problema:

1) Volver a definir la semántica SEND para hacerlo no confiable (el sistema no da garantía alguna acerca de la entrega de los mensajes).

2) Se envían reconocimientos de núcleo a núcleo (Server/Cliente) sin que estos se enteren. Así una solicitud de respuesta consta de cuatro mensajes: 1 solicitud, 2 reconocimiento, 3 respuesta, 4 reconocimiento.

3) El tercer método aprovecha la respuesta como método de reconocimiento. Si bien esto funciona bien, si la solicitud requiere de un cómputo extenso por parte del servidor, estaría mal descartar la respuesta antes de que el servidor estuviera seguro de que el cliente la haya recibido. Por esta razón a veces se utiliza un reconocimiento del núcleo del cliente al núcleo del servidor, permaneciendo éste bloqueado hasta ese entonces. Algunas veces al llegar la solicitud al servidor, si éste tarda mucho en resolverla, el servidor envía un reconocimiento antes de la respuesta.

Ejemplos:

	Con Conexión	Sin Conexión
Confiable	Transferencia de Arch. FTP	Correo Certificado (llega sino avisa)
No Confiable	Teléfono (puede haber Ruido)	Correo Simple (llega o no llega)

21.6. - IMPLEMENTACIÓN DEL MODELO CLIENTE-SERVIDOR

Existen muchas combinaciones posibles para la elección de un conjunto de primitivas de comunicación:

Elemento	Opción 1	Opción 2	Opción 3
Direccionamiento	Número de Máquina	Direcciones ralas de procesos	Búsqueda de nombres en ASCII por medio del Servidor
Bloqueo	Primitivas con bloqueo	Sin Bloqueo, copia al Núcleo	Sin Bloqueo con interrupciones
Almacenamiento en buffers	No usarlo descartar los mensajes inesperados	No usarlo mantenimiento temporario de mensajes inesperados	Buzones
Confiabilidad	No Confiable	Solicitud; Reconocimiento; Respuesta; Reconocimiento	Solicitud; Respuesta; Reconocimiento

21.7. - COMENTARIOS ACERCA DE IMPLEMENTACIÓN PROTOCOLOS Y SOFTWARE

Todas las redes tienen tamaño máximo de paquetes. Los mensajes mayores a esa cantidad, deben dividirse en varios paquetes y enviarse en forma independiente.

Uso de los reconocimientos, se pueden hacer por :

Paquete individual.

Ventaja : si se pierde un paquete, solo se retransmite el paquete;

Desventaja : se necesitan mas paquetes en la red,

o,

Reconocimiento por mensaje completo.

Ventaja : hay menos paquetes en la red;

Desventaja : Se debe retransmitir todo el mensaje en caso de pérdida.

En el modelo Cliente-Servidor, existen los siguientes tipos de mensajes que sirven para los siguiente casos:

Código	Tipo de paquete	De	A	Descripción
REQ	Solicitud	Cli	Ser	El cliente desea Servicio
REP	Respuesta	Ser	Cli	Respuesta del Servidor al Cliente
ACK	Reconocimiento	Cualq	Al Otro	El paquete anterior que ha llegado
AYA	Estas Vivo?	Cli	Ser	Verifica si el Servidor se ha descompuesto
IAA	Estoy Vivo	Ser	Cli	El Servidor no se ha descompuesto
TA	Intenta de Nuevo	Ser	Cli	El Servidor no tiene espacio
AU	Dirección desconocida	Ser	Cli	Ningún proceso utiliza esta dirección

Las secuencias más comunes son las siguientes:

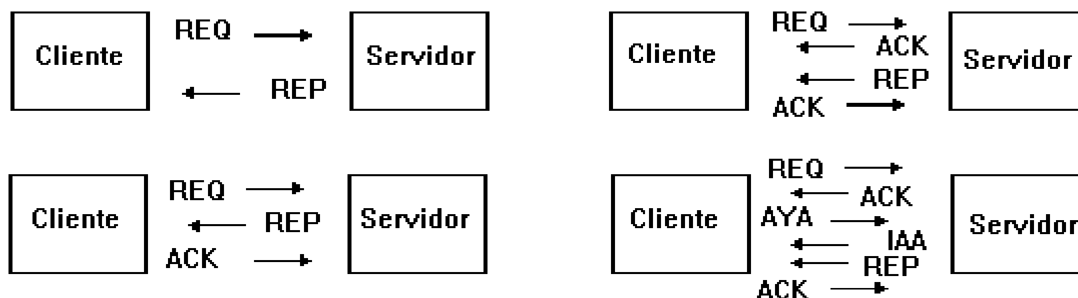


Fig. 21.1.

CAPITULO 22

REMOTE PROCEDURE CALL

22.1. - Introducción

El modelo cliente-servidor sufre de una debilidad. Todo su paradigma de comunicaciones está basado en sentencias de input/output (send/receive)

En RPC (Birrel y Nelson 1984) los procesos pueden hacer llamados a procedimientos que no residen en la máquina donde están corriendo.

La llamada a una subrutina que se encuentra en otra máquina, no se ejecuta concurrentemente, ya que el proceso llamador queda bloqueado o suspendido hasta que termine la subrutina. La información se puede transportar de un lado a otro mediante los parámetros y puede regresar en el resultado del procedimiento. Así todo pasaje de mensajes e instrucciones de I/O son invisibles al programador.

Como sabemos en la llamada a un procedimiento local, se pueden utilizar pasajes de parámetros por valor o por referencia.

Cuando la llamada es remota tiene que verse lo más local posible. Para lograr esta transparencia cuando se compila un programa, en vez de adicionar código de una librería, se une un tipo de código especial llamado client-stub (resguardo del cliente) que se va a encargar de empaquetar los parámetros, hacer un send al server y hacer un receive bloqueándose en espera de la respuesta.

Cuando el mensaje llega al server, el kernel se lo pasa al server-stub (resguardo del servidor). Para que esto ocurra, el código del server-stub tuvo que ejecutar un receive y quedarse en espera de un requerimiento. Luego desempaqueta los parámetros y llama a la rutina del server que va a procesar el requerimiento (coloca los parámetros en el Stack). Una vez resuelto le pasa los datos al stub que los empaqueta, hace un send con el mensaje empaquetado y luego hace un receive bloqueándose para recibir un nuevo mensaje.

Al recibir este mensaje el cliente, se desbloquea el client-stub, desempaqueta el mensaje y lo coloca a disposición del cliente.

La razón fundamental para la existencia de estas rutinas stubs radica en que el compilador generará en base a especificaciones formales del servidor sendas rutinas stubs, la del cliente para que empaquete los parámetros y construya el mensaje y la del servidor para que los desempaquete apropiadamente. Esto facilita la vida de los programadores, reduce la posibilidad de error y provee de transparencia al sistema respecto de las diferentes representaciones internas.

22.2. - Etapas de un RPC

1) El procedimiento cliente llama al stub cliente de manera transparente. Usando Stack.
2) El stub cliente arma el mensaje y se lo envía al kernel.
3) El kernel realiza el send del mensaje al kernel de la máquina remota.
4) El kernel remoto le da el mensaje al stub del server
5) El stub del server desempaqueta los parámetros y se los pasa al server. Usan Stack.
6) El server propiamente dicho realiza su trabajo y retorna un resultado al stub.
7) El stub del server empaqueta el valor retornado y se lo manda al kernel.

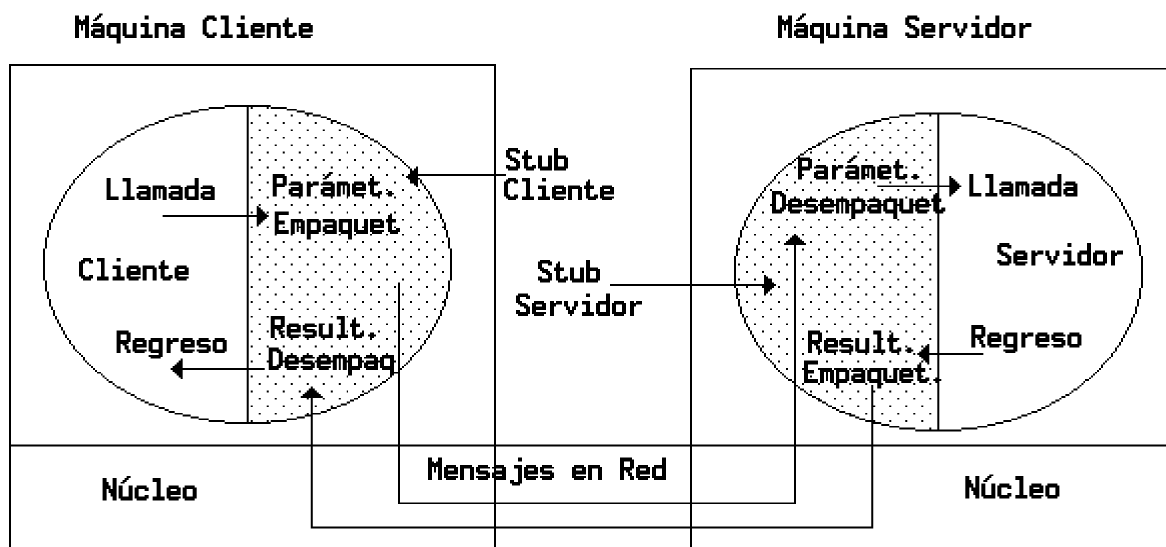


Fig. 22.1.

8) El kernel remoto envía el mensaje al kernel del cliente.
9) El kernel del cliente sube el mensaje al stub del cliente.
10) El stub cliente desempaqueta el resultado y se lo pasa al cliente.

El paso de los mensajes por los respectivos stub's es transparente tanto para el cliente como para el servidor.

Un mismo server puede proveer varias funciones, por lo tanto cada vez que llega un mensaje al stub server, debe chequearse cuál es el servicio que se requiere (que tiene que ser puesto como parte del mensaje por el stub cliente) y recién ahí realizar la llamada local.

22.3. - **Pasaje de Parámetros**

Recordemos que en forma estándar existen dos formas de pasar parámetros en una llamada a una subrutina. Uno de ellos es pasar parámetros por **valor** en cuyo caso el dato que se desea informar al procedimiento se copia directamente (usualmente se coloca en el stack) y la otra forma es por **referencia** en cuyo caso lo que se copia hacia el procedimiento es un puntero que indica la dirección en dónde se encuentra almacenada el dato en sí. Existe una tercera forma que se denomina **copia/restauración** en la cual el dato se copia como en el pasaje por valor pero el procedimiento invocado luego lo copia de regreso después de la llamada escribiendo sobre el valor original.

Veamos un ejemplo de cómo se haría este pasaje de parámetros. Supongamos que el cliente desea sumar dos números enteros (4 y 7). La llamada al procedimiento sum aparece en el cliente en donde el stub toma los dos parámetros y construye el mensaje que enviará al servidor indicando además que tipo de función solicita ya que el servidor podría proveer diferentes llamadas. Cuando el mensaje llega al servidor su stub analiza cuál es el procedimiento invocado y luego lleva a cabo la operación solicitada. Luego el stub del servidor toma el resultado de la operación lo empaqueta en un mensaje y éste se reenvía de regreso al cliente quien lo desempaqueta y se lo devuelve al proceso.

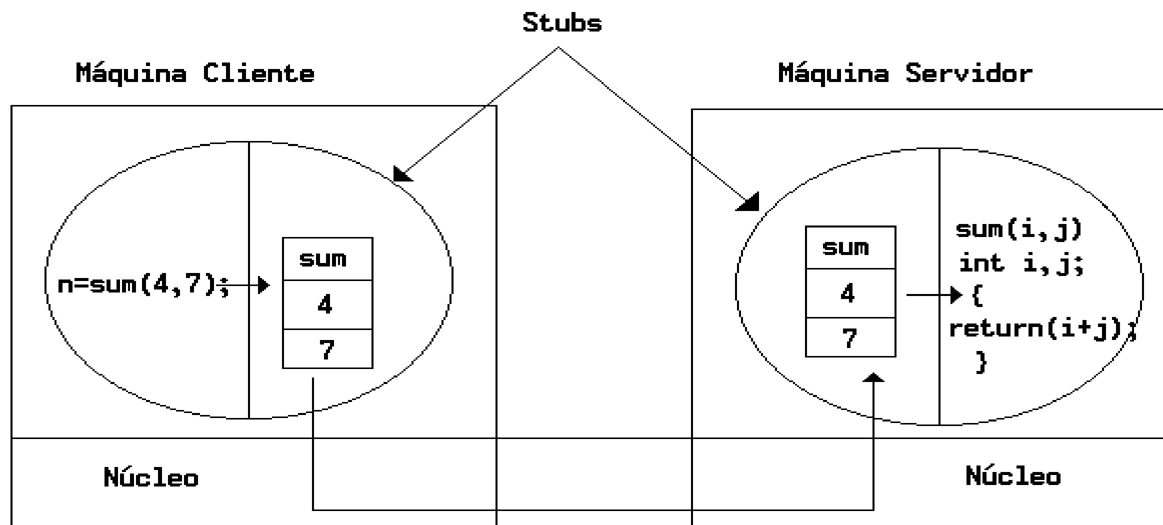


Fig. 22.2. - Cálculo remoto de sum(4,7).

Todo funciona bastante sencillamente en tanto el cliente y el servidor ejecuten en máquinas idénticas y los parámetros pasados sean de tipo escalar (enteros, caracteres, boolean). Sin embargo en un sistema distribuido de gran tamaño es común que se encuentren diferentes tipos de máquinas (Pcs, mainframes con diferente tipo de representación interna -ASCII, EBCDIC-); también se plantea un problema con los números de punto flotante y con la forma de numerar los bits en algunas máquinas (de derecha a izquierda *little endian* o de izquierda a derecha *big endian*).

Para solucionar este problema, se estableció una forma canónica para el pasaje de parámetros. Este método obliga a que el stub cliente o el servidor realicen conversiones de los mensajes recibidos o enviados siempre lo cual resulta ineficiente si ambos extremos de la comunicación proveen en forma nativa el mismo tipo de representación de datos.

Otra solución es indicar en el mensaje el tipo de máquina que lo está enviando, y no realizar ningún tipo de transformación previa. Si el receptor es el mismo tipo de máquina, toma los parámetros sin ningún inconveniente, de otra forma hace la conversión que sea necesaria.

Otro problema en el pasaje de parámetros es si se pasan punteros, ya que un puntero solo tiene sentido en el espacio de direccionamiento (address space) en donde se encuentra el proceso.
Una solución es prohibir el pasaje de punteros y parámetros por referencia.

Otra solución puede ser pasar los elementos del vector en su totalidad dentro del mensaje, así el stub del servidor podría llamar a la rutina apuntando a un buffer propio, donde guardó el vector que le pasaron como pará-

metro. Si el server escribe sobre esta parte de la memoria, el stub del servidor tiene que encargarse de devolverlo al cliente, así puede copiarlo modificado.

Luego los parámetros por referencia son reemplazados por un mecanismo de *copy/restore*.

Para optimizar más esta solución, bastaría con que el stub sepa si el buffer es un parámetro de Entrada (no necesita ser devuelto al cliente), Salida (no necesita ser enviado al servidor) o E/S.

Existe una forma de manejar el caso de apuntadores para estructuras de datos complejas. Aquí lo que se realiza es enviar el apuntador mediante la colocación en un registro y realizando un direccionamiento indirecto. La referencia inversa lo que realiza es enviar un mensaje del servidor al cliente para pedirle que le busque el dato deseado. Si bien este esquema es muy ineficiente hay ciertos sistemas que lo utilizan.

Pero aún existe un problema como ser en la función INC(i,i) [i=i+1], al stub del server le llegarían dos parámetros diferentes pero en realidad es el mismo, con lo cual si i' = 1 (como parámetro de Entrada) ... en el server ... i'' = 2 (como parámetro de Salida) [i'' = i' + 1], pero ambos hacen referencia al mismo lugar de memoria en el cliente, por lo tanto al restaurar los valores en la variable i del cliente puede quedar 1 o 2.

22.4. - Generación de un Código RPC

La generación de un código RPC se hace automáticamente. Se define un archivo donde se especifica el nombre y los tipos de parámetros de las funciones remotas al cliente, como el número de programa (o server). Esto se logra teniendo un compilador que lee las especificaciones y genera tanto el stub cliente como el stub del servidor. El código fue generado para ambos stubs a partir de la misma especificación con lo cual reduce las probabilidades de error haciendo transparente las dificultades del pasaje de términos.

22.5. - Dynamic Binding (Conexión dinámica)

Uno de los problemas existentes se basa en el direccionamiento al server, es decir la forma en que el cliente localiza al servidor.

Una forma es incluir la dirección en el código del cliente, pero esto no sería muy flexible a cambios. O sea que de existir cambios en la dirección del servidor se debería recompilar el código del cliente. Para solucionar esto se plantea el dynamic binding.

22.5.1. - Modo de operar

Lo primero que se debe hacer para poder implementar este tipo de binding, es que exista una especificación formal del servidor que incluya :

- el nombre del server,
- el número de versión, y
- una lista de procedimientos provistos por el server (tipo de parámetros que utiliza y si son de E, S o E/S).

El principal motivo de contar con esta especificación formal radica en que ésta es entrada de la rutina que generará los stubs del cliente y del servidor.

Cuando el Server comienza su ejecución, hace un call a una rutina propia de inicialización, fuera del loop principal, que se encarga de realizar un EXPORT de la interfase del server (manda un mensaje al programa BINDER (conector) indicándole que lo dé de alta y que conozca de su existencia). Para realizar la registración del server en el binder, se necesitan los siguientes datos: el nombre del server, su número de versión, un identificador único y un handle (asa - usado para localizarlo y que depende del sistema). También puede proporcionarse en esta registración información para la autenticación de mensajes (un servidor puede declarar que sus servicios sólo pueden brindarse a ciertos clientes).

Cuando el Cliente hace una llamada a un procedimiento remoto por primera vez, el stub cliente manda un mensaje al binder pidiendo que haga un import de la versión nn del server mm (Dynamic Binding puede ser gracias a un mecanismo rendez-vous y normalmente, un S.O. provee de un rendez-vous daemon sobre un port RPC fijo). El binder busca en su tabla de servers que hayan exportado una interfase que coincida con lo que el cliente está buscando. Si no existe entonces retorna un error, de lo contrario el binder retorna el handler y un identificador único (del proceso). El stub del cliente utiliza este handle como dirección a donde enviar el mensaje. El identificador único que se envía también en el mensaje, es usado por el kernel del server para direccionar automáticamente el mensaje al server correcto.

El número de versión es importante porque de esta forma el binder puede garantizar que los clientes que utilicen interfases obsoletas no puedan localizar al servidor impidiendo que realicen la llamada y obtengan resultados impredecibles.

22.5.2. - Ventajas y Desventajas

Este método de exportar e importar es muy flexible. Permite aplicar un método de autenticación a cada cliente que quiere hacer un bind con los servers.

Pero tiene el overhead extra que se genera para exportar e importar las interfases y además este binder en un sistema distribuido de gran tamaño puede convertirse en un cuello de botella.

22.6. - SEMÁNTICA DE RPC EN PRESENCIA DE FALLAS

La transparencia que proporciona RPC en cuanto a que las llamadas remotas parecen llamadas locales se ve afectada con el surgimiento de fallas, pues son difíciles de enmascarar. Existen 5 tipos de problemas que pueden ocurrir, a saber :

1) **El cliente no puede localizar al server.**
Esto provoca que el programador deba que codificar qué hacer en caso de tener un error en el bind al server. (procedimiento de EXCEPCIÓN)

2) **Se pierde el mensaje de requerimiento del cliente al server.**
Para estos casos el kernel pone un timer cuando se envía un mensaje y si no recibe un ACK dentro de un lapso, se retransmite el mensaje. Si realmente se perdió el mensaje, el server toma la retransmisión del requerimiento como la original.

3) **El mensaje de respuesta del server se pierde.**
Se depende nuevamente del timer, aunque es más difícil de implementar ya que el kernel del cliente no sabe si sucede 2) o 3) o el server está lento. Existen operaciones que son idempotentes (tales que no causan daño alguno el repetirlas o sea que no cambian su valor) como por ejemplo leer una cadena de bytes de un archivo. El problema es con las otras operaciones (por ejemplo la transferencia electrónica de fondos), y una solución para esto, es que el kernel ponga un número de secuencia en el mensaje así se podrán identificar cuál es original y cual la copia y cotejar si se ha procesado el pedido. Otra solución es ponerle a cada mensaje un header que identifique si es el original o una copia. La idea es que una solicitud original siempre puede ejecutarse inmediatamente pero las solicitudes que son copias requieren de mayor cuidado.

4) **El server se cae luego de recibir el requerimiento.**
Esto tiene un problema un poco mayor. Como ser:
a) si el server ejecutó el pedido y se cayó antes de enviar la respuesta en cuyo caso el cliente debería aplicar alguna rutina de manejo de excepciones, y
b) si el server se cayó justo antes de ejecutar el pedido, situación que el cliente debe abordar reiterando el pedido al servidor.
El núcleo del cliente no puede saber cuál de las dos situaciones ha ocurrido. Hay cuatro formas de solución:
1) *At Least Once* : (1 o más) Consiste en esperar a que el Server levante de nuevo, y mandar la operación otra vez. Esto es conveniente cuando se trata de operaciones idempotentes.
2) *At Most Once :* (0 o 1) Se da por vencido al instante y reporta un error. Asegura que el RPC se hace a lo sumo una vez o puede no completarse nunca.
3) *Exactly Once* : Es la deseable pero no hay forma de garantizarlo. Siempre va a existir un punto en donde se pierde todo rastro de lo hecho hasta el momento.
4) *Don't Know??* : Si se cae un server, el cliente no promete nada, la operación se pudo ejecutar de 0-n veces. Fácil de implementar.

5) **El cliente se cae.**
Esto implica que nadie va e estar esperando la respuesta del server quedando computaciones huérfanas (ORPHANS) que utilizan CPU, pueden lockear archivos, y utilizar recursos cuando realmente no se necesita. Además si el cliente se recupera rápidamente y se retransmitió el mensaje, puede recibir dos mensajes respuesta.
a) Una solución puede ser que el stub cliente lleve un log de los mensajes que va enviando, así cuando bootea se chequea el log y se mata explícitamente aquellos ORPHANS que hubieran quedado (EXTERMINACION). Una desventaja de esto es que hay que escribir a disco cada vez. Y puede no funcionar, ya que un RPC puede a su vez hacer otro RPC y crear GRANDORPHANS (huérfanos de huérfanos), y demás descendientes que son imposibles de localizar.
b) Otra solución se la conoce como REENCARNACIÓN que divide el tiempo en generaciones numeradas secuencialmente. O sea cuando un cliente rebootea, hace broadcast de un mensaje indicando a todas las máquinas que se inicia una nueva generación. Al recibir este mensaje, cada máquina se encarga de hacer un kill a todos aquellos procesos pertenecientes a una generación anterior. Si queda vivo algún ORPHAN, cuando llegue su reply inmediatamente se va a reconocer que es inválido ya que es de otra generación.
c) Se basa en la anterior y se llama REENCARNACIÓN SUAVE. Cuando llega un broadcast indicando una nueva generación, la máquina ubica a todas las computaciones remotas y chequea si todavía sigue existiendo su dueño. Si no lo encuentra recién ahí toma la decisión de eliminarlo.
d) Se la conoce como EXPIRACIÓN, se le da un lapso de tiempo a cada cómputo RPC para que realice su trabajo. Si este no puede terminar, tiene que pedir explícitamente que se le agrande el quantum. El problema aquí es la elección de un valor adecuado del valor del lapso de tiempo.

Todas estas soluciones no son recomendables en la práctica. La eliminación de un huérfano puede tener consecuencias imprevisibles. Por ejemplo, supongamos que un huérfano haya bloqueado uno o más registros de una base de datos, si se lo elimina súbitamente estos bloqueos pueden permanecer para siempre. Además un huérfano podría crear entradas en alguna ubicación remota de procesos que se ejecutarán en un futuro con lo cual la eliminación del padre no aseguraría la eliminación de todos sus rastros.

22.7. - **Aspectos de la implantación**

Muchas veces el éxito de un sistema distribuido descansa en su desempeño, éste a su vez depende de la velocidad de comunicación y ésta de la implantación. Veremos ahora algunos aspectos en lo cuáles las decisiones tomadas afectan la performance del sistema.

22.7.1. - **PROTOCOLOS RPC**

** La primera decisión es elegir entre un protocolo orientado a conexión o uno sin conexión:*

Conexión:

Ventajas: comunicación más fácil, el núcleo del cliente no debe preocuparse de si los mensajes se pierden o de si no hay reconocimiento.

Desventajas: En una LAN tiene pérdida de desempeño debido a que todo este software adicional estorba, además la ventaja de no perder los paquetes no tiene sentido ya que las LAN son confiables en esto.

Sin Conexión:

En general en sistemas dentro de un único edificio se utilizan protocolos sin conexión. Mientras que en redes grandes se utiliza uno orientado a conexión.

** Utilizar un protocolo estándar o alguno diseñado en forma específica para RPC, por ejemplo:*

* IP (o UDP, integrado a IP) tiene puntos a favor:

-- Ya está diseñado (ahorra un trabajo considerable)

-- Se dispone de muchas implementaciones

-- Estos paquetes se pueden enviar y recibir por casi todos los sistemas UNIX

-- Los paquetes IP o UDP se pueden transmitir en muchas de las redes existentes. En resumen IP y UDP son fáciles de utilizar, pero el lado malo es el desempeño, la cantidad de información que se agrega en el encabezado del mensaje incrementa mucho el overhead del sistema.

** Utilizar un protocolo especializado para RPC que, a diferencia de IP, o tiene que trabajar con paquetes que han estado brincando a través de la red durante unos cuantos minutos y luego aparecen de la nada en un momento inconveniente*.

Por supuesto debe ser inventado, implantado y probado lo que crea un trabajo adicional. Además el resto del mundo no se pone contento cada vez que nace un nuevo protocolo.

** Longitud del paquete y el mensaje:*

La realización de un RPC tiene un costo excesivo fijo de gran magnitud independiente de la cantidad de datos enviados, por lo que se esperaría que el protocolo y la red permitan transmisiones largas.

22.7.2. - **RECONOCIMIENTOS**

En casos que la RPC de gran tamaño tenga que dividirse en muchos paquetes es necesario determinar una estrategia de reconocimientos.

** Protocolo Detenerse y esperar (stop and wait protocol)*: establece que el cliente envíe el paquete y espere un reconocimiento antes de enviar el segundo paquete.

** Protocolo de Chorro (Blast Protocol)*: establece que el cliente mande todos los paquetes y luego espere el reconocimiento del mensaje completo. Cuando se pierde un paquete el servidor puede optar por abandonar todo, no hacer nada y esperar que el cliente haga un receso y vuelva a enviar todo el mensaje, o bien guardar lo que llegó bien y pedir que se retransmita el paquete perdido (Repetición selectiva, es buena en redes de área amplia).

Existe otra consideración más importante que es el **control de flujo**. Hay veces que el chip de interfase de red no permite recibir un número ilimitado de paquetes adyacentes debido a la capacidad del mismo. Cuando un paquete llega a un receptor que no lo puede recibir se produce un error de sobreejecución (overrun error) y el paquete se pierde (esto puede ocurrir en protocolos a chorro pero nunca en detenerse y esperar). Un emisor inteligente puede insertar un retraso entre los paquetes (esperando ocupado o iniciar un cronómetro y hacer algo mientras).

Si por otro lado la sobreejecución se debe a la capacidad finita del buffer en el chip de la red, entonces el emisor puede enviar n paquetes y después un espacio considerable, o definir un protocolo para que envíe un reconocimiento después de cada n paquetes.

Los protocolos especializados para RPC tienen un desempeño mucho mejor que los sistemas basados en IP o UDP, por un amplio margen.

Si se pierde el mensaje de reconocimiento en la práctica el servidor puede inicializar un cronómetro al enviar la respuesta y descartarla cuando llega el reconocimiento o se termina el tiempo. Además se puede interpretar una nueva solicitud del cliente como un signo de que llegó bien la respuesta.

22.7.3. - **RUTA CRITICA**

La serie de instrucciones que se ejecutan con cada RPC se llama ruta crítica la cual la podemos visualizar en la figura.

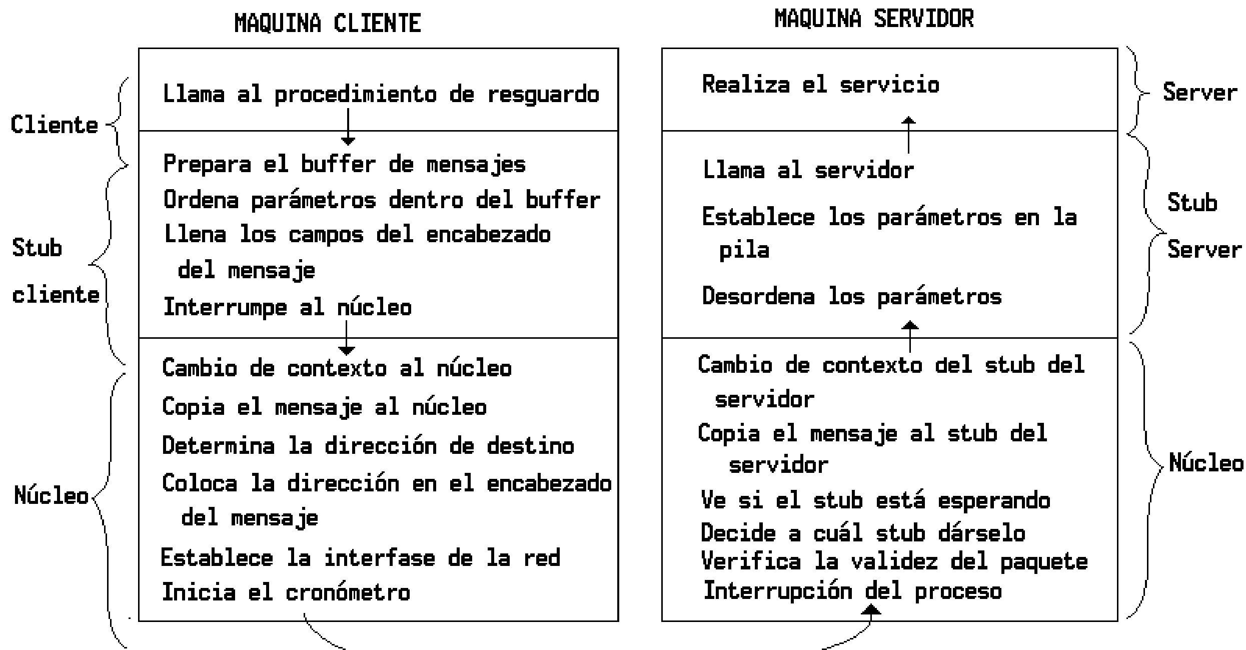


Fig. 22.3. - Ruta crítica del cliente al servidor.

Los diseñadores están interesados en determinar en qué porción de la ruta crítica se consume más tiempo. Luego de una serie de estudios y mediciones realizados los consejos que los expertos nos dan son los siguientes:

1) Evitar el uso de hardware extraño ya que si en dos etapas consecutivas de la ruta crítica intervienen componentes disímiles puede producirse una demora considerable.
2) Tampoco están contentos de haber basado su sistema en UDP debido al costo en tiempo de calcular la suma de verificación.
3) El uso de espera ocupada hubiera reducido el tiempo de cambio de contexto al espacio del usuario.

22.7.4. - **COPIADO**

El numero de veces que se debe copiar un mensaje varía de uno a ocho según el hard, soft y tipo de llamada. Hay varias técnicas para tratar de reducir esto. Una característica del hardware que es de gran ayuda es la **Dispersión-Asociación**.

Un chip de la red que realice la dispersión asociación se puede configurar de tal manera que organice un paquete mediante la concatenación de dos o más buffers de memoria. El hecho de poder asociar un paquete a partir de varias fuentes elimina el copiado. En general es más fácil eliminar el copiado en el emisor que en el receptor.

22.7.5. - **MANEJO DE CRONÓMETRO**

La cantidad de tiempo de máquina que se dedica al manejo de cronómetros no debe subestimarse. El establecimiento de un cronómetro requiere de la construcción de una estructura de datos que especifique el momento en que el cronómetro debe detenerse y la acción a realizar en caso de que eso suceda. Generalmente se usa una lista de estos procesos pendientes de que e cronómetro llegue a cero ordenada desde el de menor tiempo al mayor.

En la práctica muy pocos cronómetros ocupan todo su tiempo con lo cual el trabajo de introducir y eliminar un proceso de esta lista es un esfuerzo desperdiciado.

Además si el valor del cronómetro es muy pequeño habrá demasiadas retransmisiones y si es muy grande existirán demoras demasiado altas para la detección de la pérdida de un mensaje.

Esto sugiere que podría utilizarse una tabla de procesos, en donde una entrada contiene toda la información correspondiente a cada proceso del sistema. La activación de una RPC consta ahora de la suma de la longitud del tiempo de expiración a la hora actual y su almacenamiento en la tabla de procesos. Esto obliga a que el núcleo revise periódicamente (por ejemplo cada segundo) la tabla de procesos y si encuentra un valor distinto de cero (cero indicaría que no hay cronómetro activo para el proceso) que sea menor o igual que la hora actual entonces un cronómetro ha expirado. Los algoritmos que operan por medio de una tabla como ésta se denominan **algoritmos de barrido** (sweep algorithms).

22.7.6. - **ÁREA DE PROBLEMAS**

Hay varios problemas, algunos de ellos son:

- No se puede implantar el permiso para el acceso irrestricto de los procedimientos a las variables globales de forma remota y viceversa, a la vez que la prohibición de este acceso viola el principio de transparencia.
- Los lenguajes débilmente tipificados (por ejemplo C en donde se pueden escribir procedimientos que multipliquen dos vectores sin indicar su tamaño) traen problemas a los stubs en el ordenamiento de los parámetros.
- Transferir como parámetro un apuntador a una gráfica compleja ya que en un sistema con RPC el stub del cliente ni tiene forma de encontrar toda la gráfica.
- No siempre es posible deducir los tipos de los parámetros, ni siquiera a partir de una especificación formal del propio código. Por ejemplo la instrucción *printf* permite una cantidad arbitraria de parámetros de cualquier tipo.

CAPITULO 23

SINCRONIZACIÓN EN SISTEMAS DISTRIBUIDOS

En sistemas con una única CPU las regiones críticas, la exclusión mutua y otros problemas de sincronización son resueltos generalmente utilizando métodos como semáforos y monitores.

Estos métodos no se ajustan para ser usados en sistemas distribuidos ya que invariablemente se basan en la existencia de una memoria compartida.

23.1. - SINCRONIZACIÓN DE RELOJES

No es posible reunir toda la información sobre el sistema en un punto y que luego algún proceso la examine y tome las decisiones.

En general los algoritmos distribuidos tienen las siguientes propiedades:

1)- la información relevante está repartida entre muchas máquinas
2)- los procesos toman decisiones basadas solamente en la información local disponible
3) - debería poder evitarse un solo punto que falle en el sistema
4)- no existe un reloj común u otro tiempo global exacto

Si para asignar un recurso de E/S un proceso debe recolectar información de todos los procesos para tomar la decisión esto implica una gran carga para este proceso y su caída afectaría en mucho al sistema.

Idealmente, un sistema distribuido debería ser más confiable que las máquinas individuales.

Alcanzar la sincronización sin la centralización requiere hacer cosas en forma distinta a los sistemas operativos tradicionales.

23.1.1 - Introducción a Relojes lógicos

El timer de un computador es generalmente una máquina precisa de cristal de cuarzo.

Asociado a cada cristal existen 2 registros : Un contador y un "holding register".

El contador se decrementa con cada oscilación del cristal y cuando llega a cero produce una interrupción y se lo carga de nuevo con el valor del registro holding.

Cada interrupción es llamada un golpe de reloj (clock tick).

Con este reloj se actualiza cada segundo el reloj que está en memoria (reloj por software).

En distintas CPUs con el transcurso del tiempo los relojes de software comienzan a diferir, esto se llama clock skew (oblicuo).

Lamport en 1978 presentó un algoritmo para sincronizar relojes.

Él apuntó que la sincronización no tiene por que ser absoluta. Si dos procesos no interactúan entre sí, no es necesario que sus relojes estén sincronizados, también indicó que no es importante que los tiempos coincidan sino que concuerden en cuanto al orden en el cual ocurren los eventos.

Por convención se denominan relojes lógicos a este tipo de relojes.

Si se agrega la restricción de que además los relojes no se pueden apartar del tiempo real en más menos un cierto valor se los denomina relojes físicos.

23.1.2. - **Relojes lógicos**

Lamport define la relación "sucede antes" (happened before)

a---> b a sucede antes que b

Existen dos situaciones:

1 - si a y b son eventos en el mismo proceso y a ocurrió antes que b entonces a --> b es verdadero

2 - si a es el evento de envío de un mensaje de un proceso y b es el evento de recibir el mensaje por otro proceso, entonces a -- b es verdadero. Un mensaje no puede ser recibido antes de haber sido enviado o a la misma hora.

La relación "sucede antes" es transitiva, o sea que si :

a --> b y b --> c entonces a --> c

Pero la relación no es reflexiva pues no se da a --> a

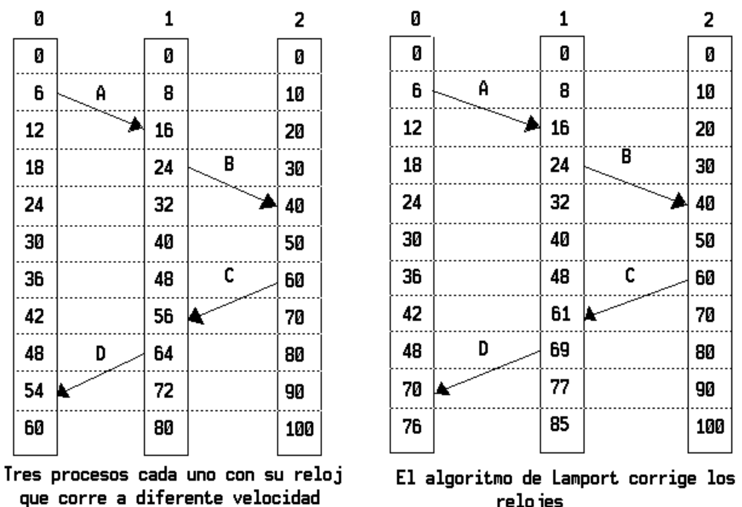


Fig. 23.1. - Algoritmo de Lamport.

Sea LC (logical clock) el valor del tiempo de un cierto evento entonces si

a -- >b entonces LC(a) < C(b)

Además el valor de LC va siempre aumentando, nunca disminuye.

Si no existe la relación --> entre 2 procesos diremos que esos dos procesos ejecutan concurrentemente.

En la Figura 23-1 se puede ver cómo los que tienen relojes más lentos los aceleran a los valores del más veloz.

Si dos eventos llegan en rápida sucesión a un proceso él debe avanzar su reloj por lo menos una vez por cada uno de ellos.

Es decir si a envía un mensaje a b y resulta que LC(a) > LC(b) se hace :

LC(b) = LC(a) + 1

Si además se agrega a la hora el número de proceso que envía el mensaje se pueden establecer las siguientes condiciones :

1 - si a sucede antes que b en el mismo proceso LC(a) < LC(b)

2 - si a y b son la emisión y recepción de un mensaje LC(a) < LC(b)

3 - para todos los eventos a y b $LC(a) \neq LC(b)$

También pueden numerarse los nodos comenzando desde 1 con lo cual es equivalente decir LC(a) a LC_1.

23.1.3. - Relojes físicos

Si bien Lamport da un algoritmo no ambiguo para ordenar eventos los valores asignados a dichos eventos no son necesariamente cercanos a la hora actual en que ocurren.

En algunos sistemas (por ejemplo los de tiempo real) la hora actual es importante. Para estos sistemas se necesitan relojes físicos externos.

Por razones de eficiencia y redundancia son deseables múltiples relojes de este tipo lo que nos lleva a los siguientes problemas :

a) cómo los sincronizamos con los del mundo real ?

b) cómo los sincronizamos entre sí ?

La hora astronómica GMT (Greenwich Mean Time) ha sido reemplazada por la hora UTC (Universal Coordinated Time) basada en los relojes de átomos de Cesio 133.

Estos relojes son tan exactos que es necesario introducir “segundos de salto” para equipararla a la hora real. En realidad la hora basada en los relojes de cesio es la hora TAI (de Tiempo Atómico Internacional) y la hora TAI corregida con los segundos de salto es la hora UTC.

La mayoría de las compañías que proporcionan la luz eléctrica basan sus relojes de 60 Hz o 50 Hz en el UTC.

Esta hora UTC es provista al público por el NIST (National Institute of Stanford Time) por medio de una estación de radio ubicada en Fort Collins Colorado, cuyas letras son WWV que envía un corto pulso al inicio de cada segundo UTC. Su exactitud es de 10 msec. También algunos satélites proveen la hora UTC.

23.1.3. - ALGORITMOS DE SINCRONIZACIÓN DE RELOJES

Si una máquina recibe la hora WWV entonces la tarea es lograr que las otras se sincronicen con ésta.

Si la hora UTC es t, el valor en la máquina p es

$$C_p(t)$$

lo ideal sería que

$$C_p(t) = t$$

es decir que

dC / dt debería ser 1

Como existen ligeras variaciones entre los ticks de los relojes se puede decir que existe una constante δ tal que

$$1 - \delta \leq C/dt \leq 1 + \delta$$

esta constante δ viene especificada por el fabricante y se denomina "tasa máxima de desplazamiento" (maximum drifting rate)

Si se desea que dos sistemas operativos no difieran en más de α entonces deberán resincronizarse por lo menos cada $\alpha / 2\delta$ segundos.

Los distintos algoritmos difieren en cómo se resincronizan.

23.1.3.1. - **Algoritmo de Cristian** (1989)

Llamamos a la máquina WWV un servidor de tiempo

Periódicamente (no más allá de $\alpha / 2\delta$ segundos) cada máquina envía un mensaje al servidor preguntándole la hora.

El servidor responde inmediatamente con su C_{UTC}.

Lo que puede hacer el receptor es setear su hora a la que recibió.

Pero existen dos problemas, uno más grave que el otro.

El grave es que el reloj del receptor no puede retrocederse. Si su reloj interno es más rápido que el del servidor setearse con C_{UTC} le traería muchos problemas.

Una forma de hacerlo es realizar las correcciones al reloj software en forma gradual.

Si el timer genera 100 interrupciones por segundo cada interrupción agrega 10 msec a la hora. Para retrasarlo la rutina de la interrupción solo agregaría 9 msec hasta llegar a la hora correcta y para adelantarlo podría sumar 11 msec.

El otro problema menor es que existe un tiempo distinto de cero que le lleva a la hora del servidor el llegar al receptor, este tiempo además empeora si el tráfico de la red es alto.

Lo que hace el algoritmo es utilizar la hora de envío del requerimiento T_0 y la hora de recepción T_1 y hace una estimación (T_1 - T_0) / 2.

Se pueden hacer mejores estimaciones si se conoce la velocidad de propagación u otras propiedades.

Para mejorar la exactitud Cristian sugirió hacer no una sino varias mediciones. Aquellas mediciones en la cuales T_1 - T_0 supere un cierto umbral son descartadas por ser víctimas de la congestión de la red y por lo tanto poco confiables.

Las restantes pueden ser promediadas para obtener un mejor valor.

Alternativamente el valor que llega más rápido puede ser tomado como el más exacto.

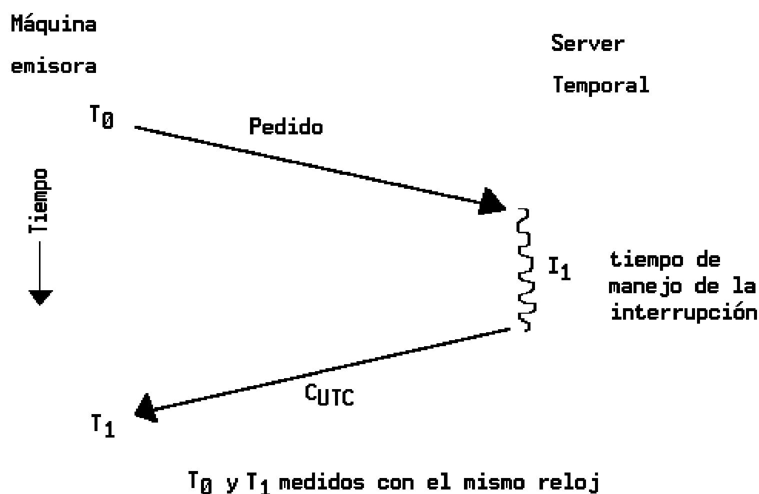


Fig. 23.2. - Obteniendo la hora actual del servidor.

23.1.3.2. - **El algoritmo Berkeley**

En el algoritmo de Cristian el servidor es pasivo. En el UNIX Berkeley ocurre exactamente lo opuesto (Gusella y Zatti 1989).

Aquí el servidor de tiempo (en realidad un demonio temporal) es activo y polea cada máquina periódicamente para preguntarles que hora tienen.

Promedia lo obtenido y de acuerdo a ello les dice que adelanten o atrasen sus relojes hasta alcanzar un cierto valor.

Este método es apto para sistemas en los que no existe ninguna máquina WWV.

La hora del demonio temporal debe ser seteada por el operador periódicamente.

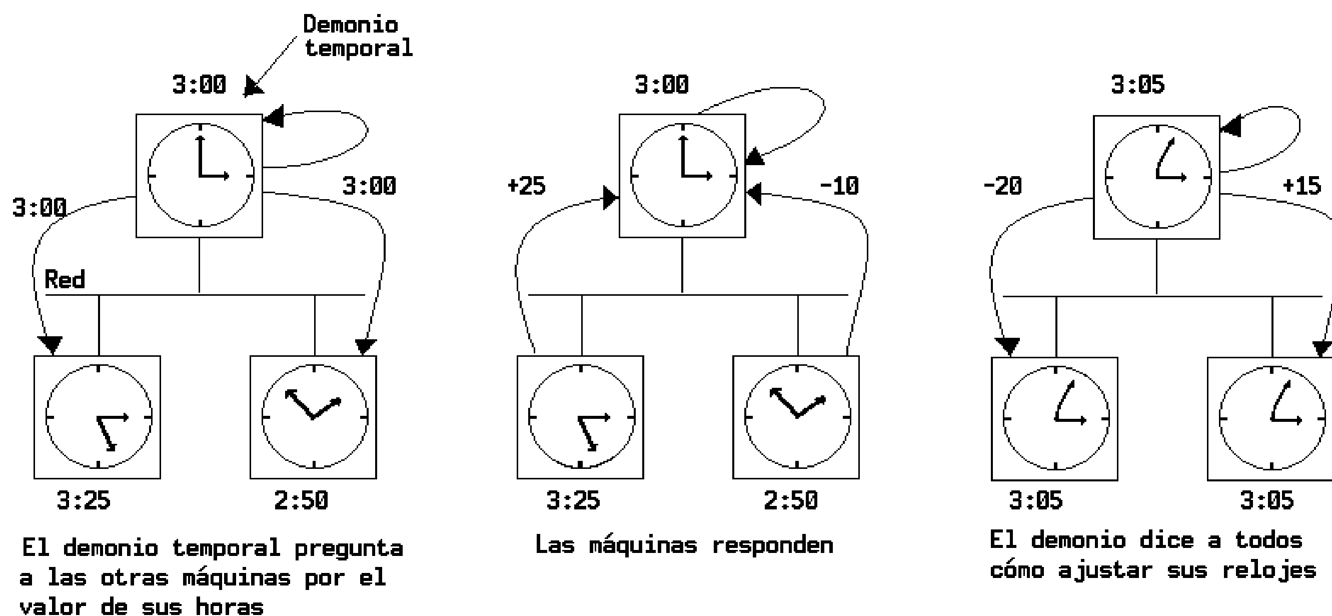


Fig. 23.3. - Secuencia del algoritmo Berkeley.

23.1.3.3. - **Algoritmos de Promedios**

Los dos métodos anteriores son centralizados con los problemas que ya conocemos.

Un algoritmo descentralizado trabaja dividiendo la hora en intervalos fijos de resincronización.
El i-ésimo intervalo empieza en el momento

$T_0 + i R$

y corre hasta

$T_0 + (i + 1) R$

donde

T_0 es un momento de acuerdo en el pasado

R es un parámetro del sistema

Al comienzo de cada intervalo cada máquina hace broadcast de su hora. Como los relojes locales difieren este broadcast no ocurre simultáneamente.

Luego de que cada máquina envía su hora empieza a correr un timer para recolectar todos los otros broadcast que lleguen durante un cierto intervalo S. Cuando todos llegan se ejecuta un algoritmo para calcular la nueva hora.

El algoritmo puede ser promediarlos a todos, que es el más simple,

También pueden descartarse los m mayores valores y los m menores y promediar el resto. Esto último sirve para liberarse de relojes que funcionen mal y pueden provocar horas sin sentido.

Otra forma es corregir cada hora que llega con un valor estimado de la propagación desde la fuente

23.1.3.4. - **Múltiples fuentes externas de horario**

Este es el caso de sistemas que necesitan exacta sincronización y cuentan con varias máquinas con hora WWV.

Cada máquina con hora UTC broadcastea su hora periódicamente, por ejemplo a cada comienzo de un minuto UTC.

Existen aquí demoras en cuanto a propagación por el medio, colisiones, congestiones de tráfico en la red, atención del receptor, etc.

Veremos por ejemplo cómo maneja esto la Open Software Foundation en su Distributed Computing Environment.

Cuando un procesador recibe todos los rangos UTC primero mira si alguno difiere de los otros. De ser así descarta a estos.

Luego hace la intersección de los valores restantes y luego calcula el punto medio y a ese valor coloca su reloj.

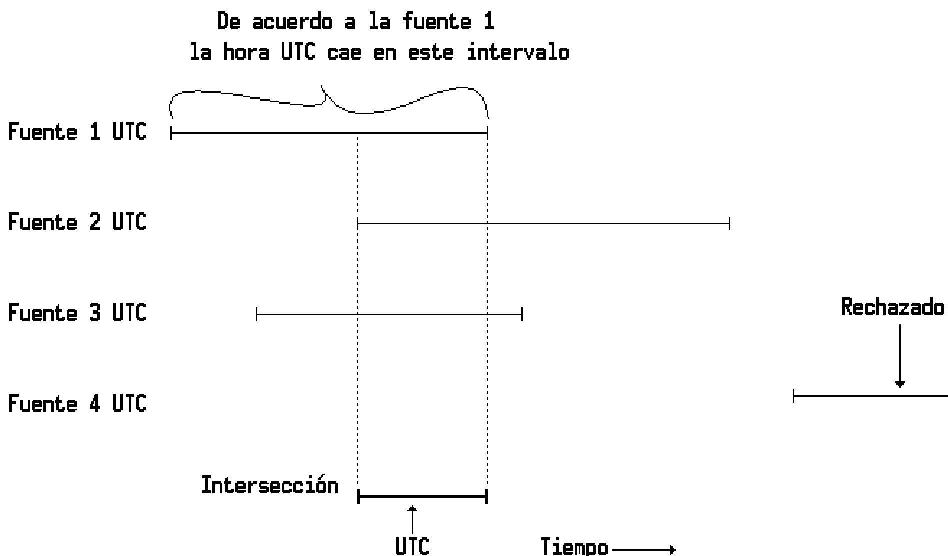


Fig. 23.4. - Calculando UTC desde múltiples fuentes externas.

Este método de sincronización es utilizado por DTS (Distributed Time Services) en DCE (Distributed Computing Environment).

DCE fue definido por la Open Software Foundation (OSF) que es un consorcio de vendedores de computadoras entre los cuales se encuentran IBM, DEC y Hewlett-Packard. No es un sistema operativo ni tampoco una aplicación, es un conjunto integrado de herramientas y servicios que pueden instalarse sobre un sistema operativo y utilizarse como plataforma para construir y ejecutar aplicaciones distribuidas. Ejemplos de sistemas operativos sobre los que puede correr DCE son AIX, SunOS, UNIX System V, HP-UX, Windows y OS/2.

En DCE los usuarios, máquinas y otros recursos se agrupan en **celdas**. En las celdas existe un servidor de tiempos que se sincroniza utilizando el algoritmo de múltiples fuentes externas. El servicio que ejecuta DCE para lograr esto se denomina DTS. Los servidores de tiempo se utilizan para sincronizar las maquinas de las celdas y también para sincronizar los servidores de tiempos entre celdas.

Una característica de DCE es que el tiempo en realidad es un intervalo de tiempo que contiene la hora exacta. Para lograr estas intersecciones de tiempo según el algoritmo se recomienda por ejemplo que cada LAN en una celda DCE tenga por lo menos 3 servidores de tiempo.

23.1.3.5. - **Otras técnicas**

Existe una forma de suavizar las diferencias de hora en forma local realizando promedios.

Supongamos un sistema distribuido en anillo o en grilla. Cada nodo puede arreglar su hora trabajando con promedios sobre las horas de sus vecinos.

23.2. - **EXCLUSIÓN MUTUA**

Para alcanzar la exclusión mutua usualmente se utilizan regiones críticas que suelen protegerse utilizando semáforos, monitores o construcciones similares.

Veremos cómo se logra la exclusión mutua y las regiones críticas en sistemas distribuidos.

23.2.1. - Un algoritmo centralizado

Se elige un proceso como coordinador. Cuando alguno desea entrar a una región crítica manda un mensaje al coordinador con su requerimiento y especificando a cuál región crítica quiere entrar.

Si nadie la está usando el coordinador responde con un mensaje de permiso.

Si alguien más desea esa región el coordinador puede no responder o responder denegando y encola el pedido.

Cuando se libera la región crítica el que la usó manda un mensaje al coordinador quien toma el primero encolado y le da el permiso.

Asegura la exclusión mutua.

Es justo ya que se atienden los pedidos en el orden que llegaron.

Nadie espera para siempre (no hay inanición, el manejo es FIFO).

El algoritmo solo requiere 3 mensajes: pedido, permiso, liberación.

Problemas:

- el coordinador es un punto de falla que haría caer al sistema
- si los procesos se bloquean luego de que les deniegan el permiso no pueden darse cuenta de si el coordinador se murió.
- el coordinador es un cuello de botella

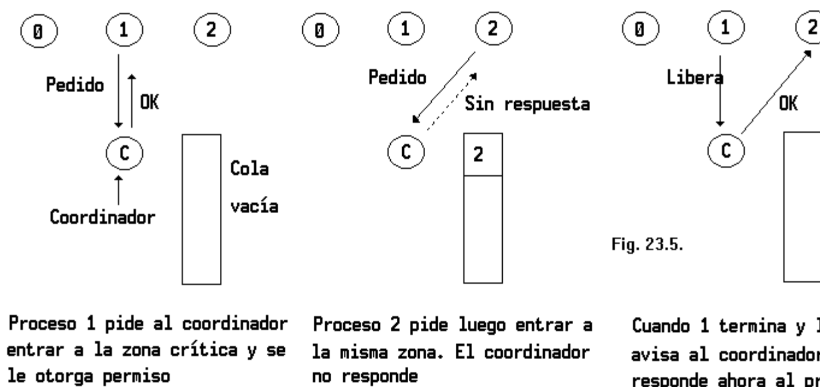


Fig. 23.5.

23.2.2. - Un algoritmo distribuido

Luego del trabajo de Lamport (1978), Ricart y Agrawala (1981) presentaron un trabajo más eficiente.

Su algoritmo requiere que todos los eventos en el sistema estén ordenados.

Cuando un proceso desea usar una zona crítica construye un mensaje formado por :

- la región que quiere usar
- su número de proceso
- su hora

Y lo envía a todos los procesos incluso él mismo.

Se supone que el envío de mensajes es confiable o sea que todos llegan.

Cuando le llega a un proceso, depende lo que hace según el estado en que se encuentre respecto de la región crítica solicitada :

1- si el receptor no está en esa zona crítica ni desea usarla le envía un mensaje de OK

2- si el receptor está usando esa zona pero todavía no responde y encola el pedido.

3- si el receptor desea usar la zona crítica pero todavía no entró compara la hora del mensaje con la hora de su propio mensaje enviado. La menor hora gana. Si el que llegó es menor el receptor le manda un OK, caso contrario no responde y encola el pedido.

El emisor una vez que todos le dieron permiso entra a la zona crítica y cuando la libera manda un OK a todos.

Si dos procesos envían un pedido en forma simultánea (iguales valores de LC) toma el recurso el que tenga el "orden" más bajo (ver fig. 23.6).

Este algoritmo asegura la exclusión mutua y no tiene inanición.

Se requieren 2 (n - 1) mensajes enviados siendo n la cantidad de procesos en el sistema.

No existe un único punto de falla.

Lamentablemente el único punto de falla fue reemplazado por n puntos de falla ya que si un proceso se cae no responderá a los requerimientos y parará al sistema.

Esto se arregla si se obliga al receptor a responder siempre ya sea permitiendo o denegando.

El emisor puede, al perder una respuesta, esperar un time out, repetirla y llegar a la conclusión de que un receptor está muerto.

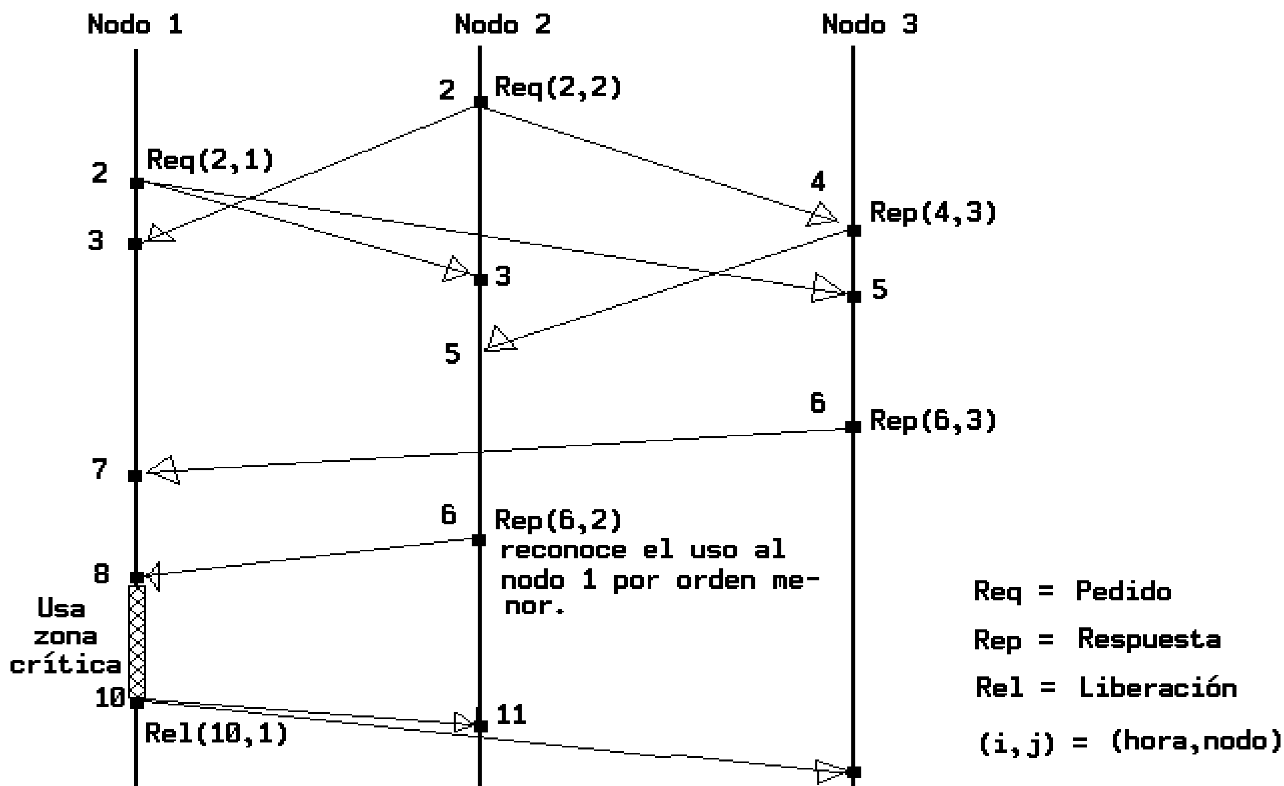


Fig. 23.6. - Sincronización distribuida.

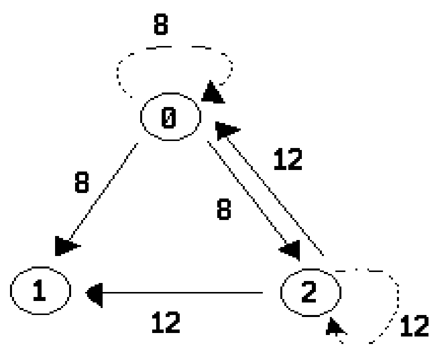
Otro problema es que o se utilizan primitivas de comunicación de grupos o cada proceso debe llevar la lista de los miembros del grupo incluyendo los nuevos procesos o los que se van o se caen.

En oposición al cuello de botella del algoritmo centralizado, aquí todos los procesos intervienen en todas las decisiones que implican uso de zonas críticas.

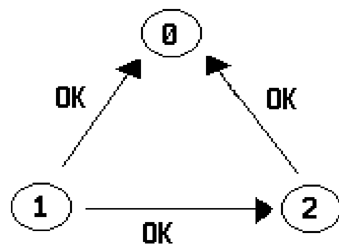
Se pueden hacer algunas mejoras, por ejemplo, que un proceso entre a zona crítica cuando recolectó permiso de la simple mayoría de procesos y no de todos. En este caso el que dio permiso a uno no puede darlo a otro hasta que el primero haya liberado.

23.2.3. - Un algoritmo token ring

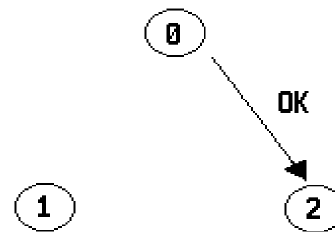
Fig. 23.7.



Dos procesos quieren la misma región crítica al mismo tiempo



El proceso 0 tiene la menor marca temporal y gana



Cuando 0 termina envía un OK entonces el 2 puede entrar a la zona crítica

Cada proceso tiene una posición asociada en el anillo y sabe cuál es su sucesor y su antecesor.

Pasa un token de uno a otro. Cuando un proceso adquiere el token entra a la región crítica, hace su trabajo, sale y restaura el token en el anillo.

Asegura exclusión mutua, no hay inanición.

Problemas :

*) pérdida del token, que deberá ser regenerado. La detección de esta pérdida no es fácil ya que si hace una hora que no aparece puede ser que lo esté usando alguien.
*) si un proceso se cae. Es más fácil de recuperar que en los casos anteriores ya que el vecino que trata de pasarle el token puede detectar que está muerto. Todo esto requiere que cada proceso mantenga la configuración actual del anillo.

23.2.4. - Comparación de los tres algoritmos

ALGORITMO	Mensajes por entrada/salida	Demora antes de entrar (en cantidad de mensajes)	PROBLEMAS
Centralizado	3	2	Caída del coordinador
Distribuido	2 (n - 1)	2 (n - 1)	Caída de cualquier proceso
Token Ring	1 a ∞	0 a n - 1	Pérdida del token, proceso caído

Es irónico que el algoritmo distribuido es aún más sensitivo a las caídas que el centralizado.

23.2.5. – Usos de relojes: Consistencia de cache

Es deseable en un sistema distribuido que los clientes puedan ocultar (caching) archivos en sus estaciones de trabajo. Esto provoca un problema de inconsistencia si dos clientes actualizan el archivo al mismo tiempo.

La solución usual es diferenciar si el archivo se oculta para lectura o para escritura, no obstante si un cliente desea escribir un archivo que ya fue obtenido anteriormente para lectura por otro cliente, el servidor debe invalidar la copia de lectura aún cuando sea una copia realizada hace varias horas.

Esto se puede solucionar utilzando relojes. Cuando el cliente obtiene un archivo se le otorga una especie de **renta** en la cual se especifica el tiempo de validez de la copia. Cuando la renta está a punto de expirar el cliente puede solicitar su revalidación. Si la renta expira la copia no puede utilizarse pero si el cliente aún desea usarla puede solicitar al servidor una nueva renta con lo cual el servidor (si el archivo es aún válido) genera la nueva renta y se la envía al cliente sin tener que retransmitir el archivo.

Si otro cliente desea utilizar un archivo para escritura el servidor debe notificar a los clientes que lo tienen en lectura y cuya renta no haya expirado, que la invaliden. Si un cliente está caído, o el link está caído a ese cliente, el servidor puede simplemente esperar hasta que la renta expire y otorgarle la escritura al que la requirió.

En el esquema tradicional sin relojes en donde el permiso debe devolverse explícitamente al servidor, el problema de la caída del cliente es complicado para el servidor ya que no sabe si el cliente se cayó o si está lento o si se cayó el link de conexión.

23.3. - **DETECCION DE FALLAS - ALGORITMOS DE ELECCIÓN**

En sistemas distribuidos es difícil diferenciar una falla de conexión, de procesador o pérdida de mensaje.

Si ambos nodos tienen una conexión física directa es posible determinar si la falla de conexión o caída de procesador ha ocurrido enviando un mensaje y no recibiendo respuesta y además alcanzando un "time-out".

Algunas de las fallas típicas que pueden producirse son :

1)- caída de un proceso en un nodo
2)- conexión física (si existe) caída
3)- caminos alternativos caídos
4)- mensaje perdido (regeneración de tokens en algunos protocolos)
5)- duplicidad de tokens
6) fracaso al transferir el token

Si se ha producido una falla y ésta fue detectada puede hacerse:

- buscar caminos alternativos
- si el nodo caído era un coordinador hay que buscar uno nuevo

La búsqueda de un nuevo coordinador requiere de algún algoritmo.

Este coordinador debe su existencia a la necesidad de que un proceso actúe como iniciador, secuenciador o para hacer algo en especial. No importa cuál pero alguien debe hacerlo.

Veremos algoritmos de elección de este coordinador. Si todos los procesos son iguales, sin características distintivas no hay forma de seleccionar uno en especial.

Asumimos que cada proceso tiene un único número, por ejemplo, dirección del nodo en la red, y que existe un único proceso por máquina.

Los algoritmos tratan de ubicar el proceso con mayor número y designarlo como coordinador. Difieren en cómo lo hacen.

Se asume que cada proceso conoce el número de los otros procesos, lo que no sabe es si están o no activos.

El objetivo es que cuando se inicia el algoritmo el mismo termina con un acuerdo generalizado sobre quién va a ser el nuevo coordinador.

23.3.1. - Algoritmo Bully (García - Molina 1982)

Cuando el coordinador no está respondiendo a los requerimientos, un proceso P hace :

1) P envía un mensaje de ELECCIÓN a cada proceso con número mayor a él
2) si nadie responde, P gana la elección y se convierte en el coordinador
3) si alguien responde éste toma el control, el trabajo de P termina

Un proceso puede recibir un mensaje de ELECCIÓN de los procesos de menor número que él, entonces le envía al emisor un OK para indicar que está vivo y se hará cargo.

A la larga solo queda uno que no recibe el OK y se autoelige coordinador y envía una señal de COORDINADOR a los otros nodos.

Si un coordinador caído se restaura envía un mensaje de COORDINADOR a los otros nodos obligándolos a someterse a él.

23.3.2. - Un algoritmo anillo

Este es otro algoritmo basado en un anillo pero a diferencia del anterior no existe un token.

Cuando uno detecta que el coordinador ya no funciona construye un mensaje de ELECCIÓN con su número de proceso y enviándoselo a su sucesor, si el sucesor está caído el emisor lo saltea al próximo.

Por cada proceso que pasa éste le agrega su número al mensaje. Eventualmente vuelve al emisor original el que lo detecta por que ve su propio número en el mensaje.

Entonces cambia el mensaje por un mensaje de COORDINADOR y lo hace circular de nuevo para informar :

- quién es el nuevo coordinador (el proceso de mayor número)
- quienes son los miembros del nuevo anillo.

Después de la vuelta de este mensaje todos vuelven a su trabajo.

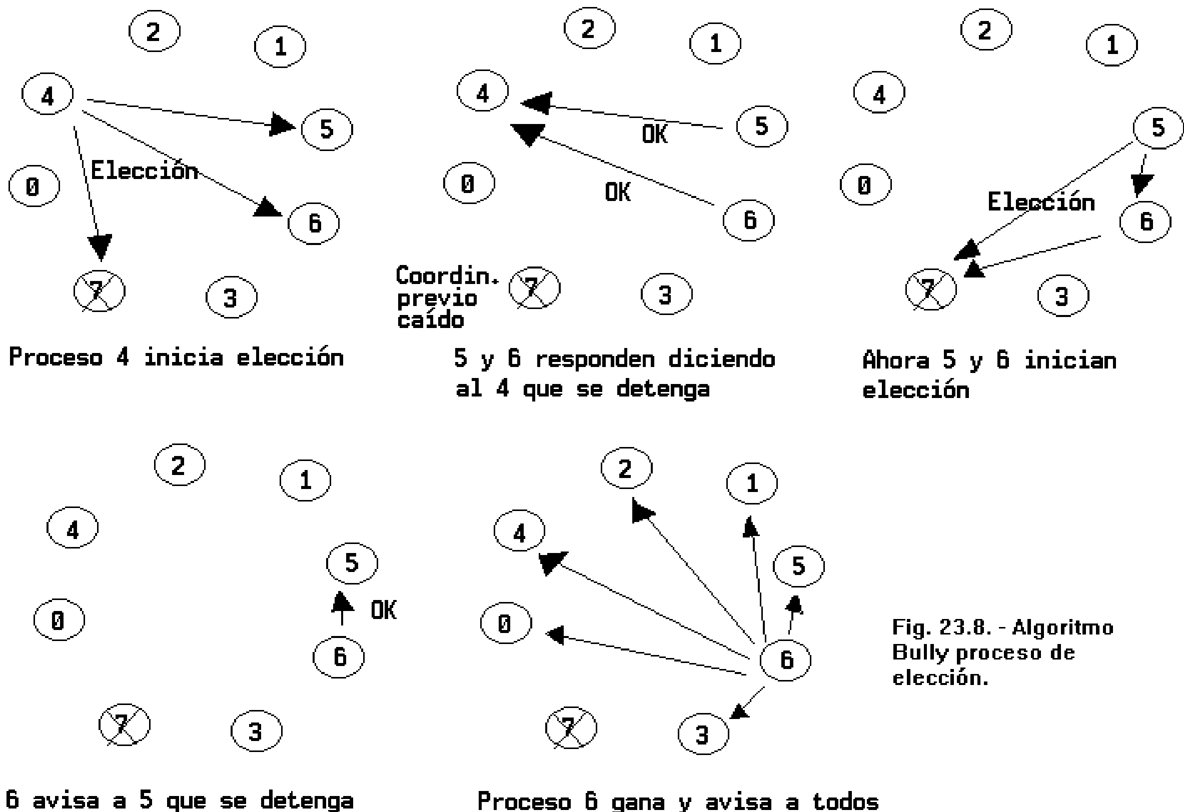


Fig. 23.8. - Algoritmo Bully proceso de elección.

23.3.3. - Tokens duplicados

En este caso si un nodo que posee el token escucha en la línea que algún otro lo posee automáticamente se inhibe cediendo el derecho al otro. Si el otro nodo a su vez también se inhibe algún nodo regenerará nuevamente el token.

23.3.4. - Falla al transferir el token

Uno de los casos de falla de transferencia del token es cuando el nodo luego de transmitir escucha solamente su propia trama y luego silencio. Primero reintenta la transmisión y si la situación se repite supone en primera instancia que el receptor se ha caído y envía por lo tanto un mensaje pidiendo quién es el nodo siguiente del anillo. Si recibe respuesta del sucesor actualiza su información y transmite. Si por último no recibe respuesta envía un mensaje solicitando su sucesor en el anillo a lo cual todos los nodos están obligados a responder y el nodo emisor resuelve entonces cuál es el nodo al cual deberá transmitir. Si esto último falla entonces todos los nodos han dejado el anillo o falla la conexión o la recepción del nodo emisor falla en cualquiera de los casos el nodo emisor pasa a un estado inactivo y transmitirá solamente cuando vuelva a recibir un nuevo token.

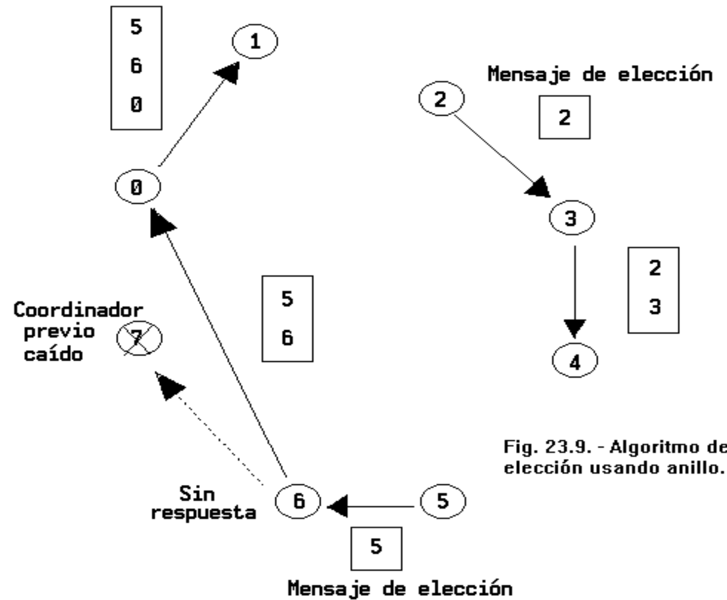


Fig. 23.9. - Algoritmo de elección usando anillo.

23.4. - **TRANSACCIONES ATÓMICAS**

Las técnicas de sincronización utilizadas son esencialmente de bajo nivel, como los semáforos.

Para esto hay que conocer detalles de la exclusión mutua, administración de regiones críticas, prevención de abrazo mortal y recuperación de caídas.

Deseamos un mayor nivel de abstracción el cual denominaremos "transacciones atómicas" o simplemente "transacciones"

23.4.1. - Introducción a transacciones atómicas

El modelo original proviene del mundo de los negocios en el cual para cerrar un trato es necesario una serie de acuerdos entre los negociantes. En computación sucede algo similar.

Un proceso anuncia que desea iniciar una transacción con uno o más otros procesos. Se pueden negociar varias opciones por ejemplo, crear y borrar objetos, y hacer algunas operaciones mientras tanto.

Luego el iniciador anuncia que desea que todos los otros completen su trabajo iniciado antes. Si todos ellos están de acuerdo, los resultados quedan en forma permanente. Si uno o más rehusa (o se cae antes de estar de acuerdo), la situación se revierte al estado exacto anterior al inicio de la transacción con todos los efectos laterales sobre objetos, archivos, bases de datos, etc., eliminados mágicamente.

Esta propiedad de todo-o-nada facilita la tarea del programador.

Este concepto proviene de los viejos sistemas de los años 60 en los cuales todavía no existían discos y lo que se hacía con cintas era por ejemplo lo que se ve en la figura 23-10.

Estos viejos sistemas tenían la propiedad de todo-o-nada de las transacciones atómicas.

Hoy en día por ejemplo una operación de transferencia de fondos de una cuenta bancaria a otra consta de 2 operaciones :

1 - extracción de cuenta A por x $

2 - depósito en cuenta B de x $

Si se corta la operatoria entre 1 y 2 los saldos no serían correctos a menos que se pudiera revertir la operación 1.

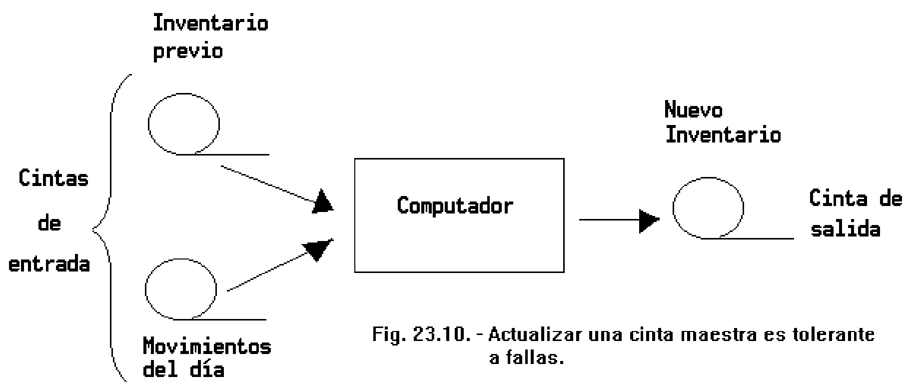


Fig. 23.10. - Actualizar una cinta maestra es tolerante a fallas.

23.4.2. - El modelo de transacción

Vamos a desarrollar un modelo de cómo es una transacción y qué propiedades posee. Asumimos lo siguiente :

- existen procesos independientes y cualquiera de ellos puede fallar
- la comunicación no es confiable en el sentido de que pueden perderse mensajes pero a niveles inferiores se usa time out y protocolo de retransmisión para recuperar estas fallas.
- asumimos que los errores de comunicación son manejados en forma transparente por el software subyacente

23.4.2.1. - **Almacenamiento estable**

La RAM pierde información cuando se corta la corriente. El disco sobrevive pero si aterrizan las cabezas, chau.

El almacenamiento estable está diseñado para sobrevivir a casi todo excepto catástrofes mayores como terremotos, etc.

Se implementa con un par de discos, uno de ellos es el reflejo exacto del contenido en el otro.

Sean los discos 1 y 2.

Primero se graba un bloque en el disco 1 y se lo verifica y luego se graba el bloque en el disco 2.

Si el sistema se cae antes de grabar el disco 2, se comparan ambos discos (al reiniciar actividad) y en las diferencias se da por bueno el disco 1 y se actualiza así el 2.

Si un bloque se daña por causas naturales (por ejemplo polvo) se lo puede regenerar a partir del mismo bloque en el otro disco.

Este tipo de almacenamiento lo hace sumamente útil para las transacciones atómicas ya que se baja a posibilidades muy pequeñas la probabilidad de errores.

23.4.2.2. - **Primitivas de Transacción**

Estas primitivas deberían estar provistas por el sistema operativo que trabaje con transacciones. He aquí algunos ejemplos básicos :

BEGIN_TRANSACTION : los comandos que siguen forman la transacción
END_TRANSACTION : fin de la transacción e intento de completar el trabajo (COMMIT)
ABORT_TRANSACTION : matar la transacción y restaurar al estado previo
READ : leer datos de un archivo u otro objeto
WRITE : grabar datos en un archivo u otro objeto.

La lista varía según la clase de objetos que se utilicen en una transacción. Por ejemplo si es un sistema de correo electrónico existirán primitivas para enviar (SEND) o recibir (RECEIVE) mensajes.

El BEGIN y el END se usan para delimitar el alcance de la transacción. Por ejemplo :

```
BEGIN_TRANSACTION
        reservar BUE - MIA
        reservar MIA - ORL
        reservar ORL _ DALLAS -----------> si este falla toda la transacción se aborta
END_TRANSACTION
```

23.4.2.3. - **Propiedades de las transacciones**

Tienen 4 propiedades esenciales

1)- SERIALICIDAD : las transacciones concurrentes no interfieren entre sí

2)- ATOMICIDAD : para el mundo exterior la transacción aparece como indivisible

3)- PERMANENCIA : una vez que una transacción hizo COMMIT los cambios son permanentes

4)- CONSISTENTES : la transacción mantiene los invariantes del sistema.

La primer propiedad asegura que si o más transacciones se ejecutan al mismo tiempo para cada una de ellas o para las otras el resultado final puede verse como si todas ellas se hubieran corrido secuencialmente en algún orden (ver tabla).

BEGIN_TRANSACTION	BEGIN_TRANSACTION	BEGIN_TRANSACTION
x = 0	x = 0	x = 0
x = x + 1	x = x + 2	x = x + 3
END_TRANSACTION	END_TRANSACTION	END_TRANSACTION

Itinerario 1	x = 0	x=x+1	x = 0	x=x+2	x = 0	x=x+3	Legal
Itinerario 2	x = 0	x = 0	x=x+1	x=x+2	x = 0	x=x+3	Legal
Itinerario 3	x = 0	x = 0	x=x+1	x = 0	x=x+2	x=x+3	Ilegal

Es parte del sistema el asegurar que las operaciones individuales son correctamente intercaladas.

La segunda propiedad es la que ya vimos del todo-o-nada. Cuando una transacción se está ejecutando el resto de los procesos del sistema sean o no transacciones no puede ver nada de los estados intermedios de la transacción.

La tercer propiedad se refiere al hecho de que una vez que se hizo el COMMIT no hay forma de volver las cosas atrás.

La cuarta propiedad puede verse claramente en los sistemas electrónicos de transferencias de fondos en los que se mantiene la conservación del dinero.

23.4.2.4. - **Transacciones anidadas**

Las transacciones pueden contener subtransacciones llamadas transacciones anidadas. La transacción madre puede hacer fork de hijos para que corran en paralelo y así siguiendo.

Existe un problema importante. Si existen subtransacciones en paralelo y una de ellas hace COMMIT y su padre aborta volviendo todo a su estado anterior esto iría en contra de la propiedad de permanencia. Sin embargo el significado es claro.

Cada subtransacción que se inicia obtiene su propia copia a su espacio de trabajo privado para hacer lo que desea. Si ella hace COMMIT su universo reemplaza al del padre. Si luego inicia otra subtransacción ve los resultados de la primera. Es decir que el COMMIT de la subtransacción solo afecta el universo del ancestro y no el mundo real.

23.4.3. - Implementación

Esta claro que si la transacción actualiza todo tipo de objetos esto no puede hacerse atómicamente y la restauración tampoco es simple. Veremos dos métodos de implementación.

23.4.3.1. - **Espacio de trabajo privado**

La transacción obtiene una copia de toda la información que necesita y trabaja sobre ella en un espacio propio privado.

Problema: costo del copiado, aunque se pueden hacer algunas mejoras. Por ejemplo en los casos en que solamente lee no se realiza copia de esta información en el espacio privado (amenos que haya cambiado desde el inicio de la TR).

Por ejemplo cuando empieza un proceso se le asigna un espacio privado vacío, cuando abre un archivo para leer el puntero lleva al archivo real. Si se trata de una subtransacción el puntero lleva al espacio del padre y de

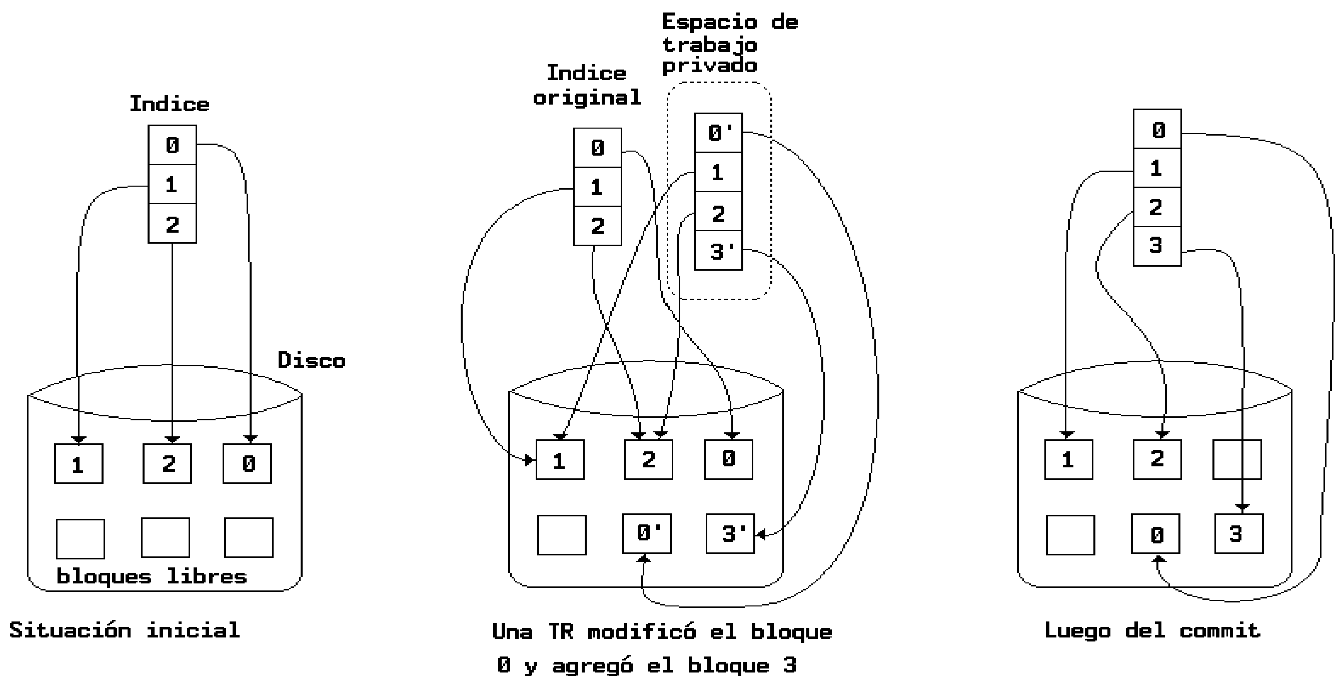


Fig. 23.11. - Actualización en espacio privado de trabajo.

allí al archivo real.

Cuando se abre un archivo para grabar los punteros desde las subtransacciones llevan al espacio privado de la TR madre y entonces allí sí se copia la información.

Una segunda mejora introduce que en lugar de copiarse el archivo entero solo se copia el archivo de índices del archivo.

En la figura 23.11 vemos por ejemplo como la TR ve solo lo que ella está modificando.

Cuando la TR aborta, su espacio privado se deletea y se devuelven los bloques libres a la lista.

Si se cumple, se reemplaza el índice en forma atómica y se actualiza la lista de espacios libres.

23.4.3.2. - **Log de Grabación Adelantada**

Es otro método también llamado Lista de Intenciones. Con este método el archivo es grabado en su lugar original pero antes de cambiar cualquier bloque se graba un registro en este log en almacenamiento estable que indica :

- qué TR está haciendo el cambio
- qué archivo y bloque se cambia
- cuál es el viejo valor
- cuál es el nuevo valor

x = 0 ;			
y = 0 ;			
BEGIN_TRANSACTION	Log	Log	Log
x = x + 1 ;	x = 0 / 1	x = 0 / 1	x = 0 / 1
y = y + 2		y = 0 / 2	y = 0 / 2
x = y * y			x = 1 / 4
END_TRANSACTION			
La transacción	Resultado en el log después de ejecutar cada sentencia		

Una vez que el log se grabó entonces se graba el archivo.

Si la TR hace COMMIT entonces se graba un registro en el log y no es necesario tocar los archivos porque ya están actualizados.

Si la TR aborta él los usa para retrotraer todo al estado original. Esta acción se llama ROLLBACK.

El log sirve también para recupero en casos de caídas ya que permite retroceder en la TR o verificar que lo último realizado por la TR efectivamente llegó a grabarse en el archivo.

23.4.3.3. - **Protocolo commit de dos fases**

En un sistema distribuido el COMMIT puede requerir de la cooperación de múltiples procesos en distintas máquinas.

Este protocolo fue ideado por Gray en 1978, si bien no es el único es el más usado en la actualidad.

Uno de los procesos involucrados funciona como coordinador, usualmente el que está ejecutando la TR.

El protocolo se inicia cuando el coordinador graba una entrada en el log diciendo que está empezando el protocolo COMMIT y enviando a los otros procesos un mensaje de que se preparen para el COMMIT.

Cuando un subordinado recibe el mensaje mira a ver si está listo para el COMMIT, hace una entrada en el log y devuelve su decisión al coordinador.

Cuando el coordinador recibió todas las respuestas sabe si puede o no hacer el COMMIT.

Si uno o más no pueden hacer COMMIT o no responden la TR se aborta.

El coordinador entonces graba su decisión en el log y se la informa a sus subordinados.

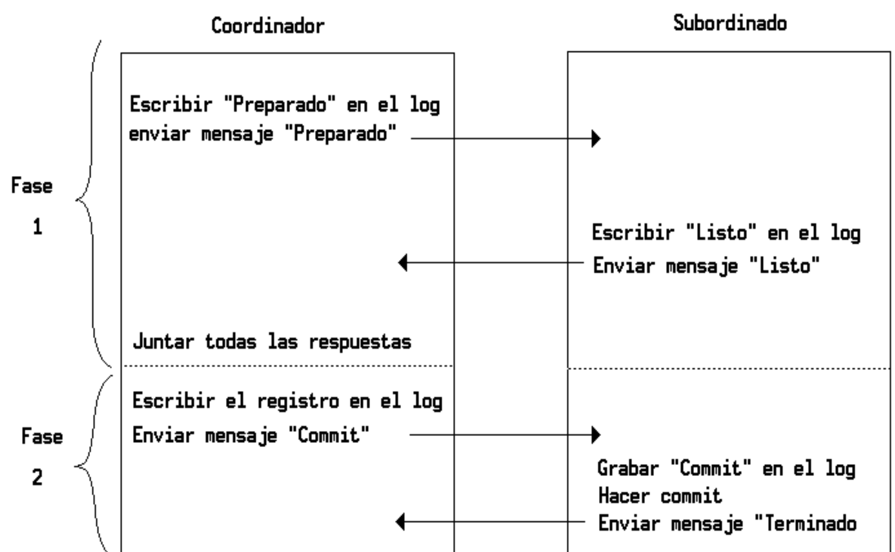


Fig. 23.12. - Protocolo Two-Phase Commit.

Esta grabación en el log es la que vale no importa lo que pase después.

Ya que se usa almacenamiento estable para este log este protocolo es muy flexible a múltiples caídas.

Si el coordinador se cae al mandar el primer mensaje cuando se recupera sigue donde estaba y vuelve a repetir el mensaje.

Si el coordinador se cae después de grabar el resultado de todas las respuestas cuando se recupera solo tiene que reinformar a sus subordinados.

Si el subordinado se cae antes de responder el primer mensaje el coordinador le enviará mensajes hasta que se recupere.

Si se cae después, cuando se recupere verá en el log que es lo que sucedió y actuará según ello.

23.4.4. - Control de concurrencia

Necesitamos mecanismos que controlen el acceso concurrente. Veremos tres algoritmos.

23.4.4.1. - **Bloqueo (locking)**

Cuando un proceso quiere usar un archivo lo bloquea. Puede existir utilizando un administrador centralizado de bloqueos o administradores locales de bloqueo para los archivos locales.

El bloqueo se adquiere y libera a través del sistema de transacciones, no se necesita ninguna acción del programador.

Pueden existir bloqueos de :

- lectura : permiten otras lecturas (compartidos)
- grabación : no permiten otras lecturas (excluyentes)

El bloqueo puede ser a más bajo nivel, por ejemplo un registro, o a mayor nivel, por ejemplo una base de datos.

Esta cuestión del grado del bloqueo se llama GRANULARIDAD DEL BLOQUEO.

Cuanto más bajo el grado, mayor paralelismo se puede lograr, pero se necesitan más indicadores de bloqueo, es más caro y puede llevar más a abrazos mortales.

El adquirir el bloqueo solo cuando es necesario y liberarlo ni bien no se lo necesita puede llevar a inconsistencias y abrazo mortal.

Por otra parte muchas TR están implementadas con un uso de bloqueo que se llama BLOQUEO DE DOS ETAPAS.

En la primer etapa la TR adquiere todos los bloqueos (fase de crecimiento) y luego los libera durante la fase de tracción (segunda etapa).

Si el proceso no llega a actualizar algún archivo antes de llegar a la fase de retracción entonces se puede manejar la falla de adquirir algún bloqueo simplemente liberándolos a todos esperando un poco y empezando todo de nuevo.

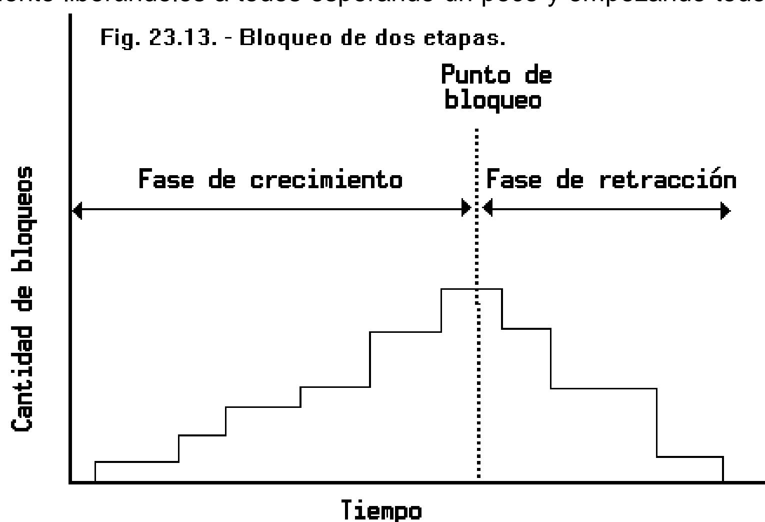


Fig. 23.13. - Bloqueo de dos etapas.

Más aún, se prueba que si todas las transacciones utilizan este mecanismo de dos etapas todas las formas posibles de intercalar las transacciones son serializables.

La fase de retracción en muchos sistemas ocurre cuando la TR terminó bien (COMMIT) o mal (ABORT), esto se llama BLOQUEO DE DOS ETAPAS ESTRICTO, y tiene dos ventajas :

1) una TR siempre lee un dato generado por una TR que ya terminó con lo cual los cálculos propios no van a fallar por no poder leer un dato.
2) las adquisiciones y liberaciones pueden ser manejadas por el sistema sin que la TR se preocupe por ello. Esta estrategia hace desaparecer la eliminación en cascada (cascaded aborts) que se produce cuando hay que deshacer una TR completada (COMMIT) debido a que vio un archivo que no debería haber visto.

Puede haber abrazo mortal si dos TR piden recursos en orden opuesto. Se usan técnicas clásicas para evitar esto, como ser que se pidan los bloqueos en un cierto orden canónico preestablecido (evita hold-&-wait).

También se pueden detectar ciclos teniendo la información de qué procesos tienen cuáles bloqueos y cuáles desean adquirir.

También puede existir un timeout sobre el tiempo que un proceso puede mantener un cierto bloqueo.

23.4.4.2. - **Control de concurrencia optimista** (Kung y Robinson 1981)

La idea es hacer lo que se desea y no poner atención a lo que los otros hacen. Si existe algún problema ocúpese de él después.

Lo que hace este esquema es llevar un registro de cuáles archivos han sido leídos y escritos y cuando una TR pide COMMIT revisa si sus archivos fueron cambiados por algún otro, de ser así aborta la TR sino la cumple.

Esta técnica anda bien en las implementaciones basadas en espacios privados de trabajo donde cada TR altera sus archivos en forma privada.

Ventajas: está libre de deadlock y permite máximo paralelismo ya que nadie espera para realizar un bloqueo.

Desventaja: es que al fallar la TR debe reejecutarse y si existe mucha carga la probabilidad de fallas crece haciendo que este esquema resulte una pobre elección.

23.4.4.3. - **Sellos Temporales** (timestamps Reed 1983)

Se asigna a cada TR un sello de tiempo en el momento en que se realiza el BEGIN_TRANSACTION.

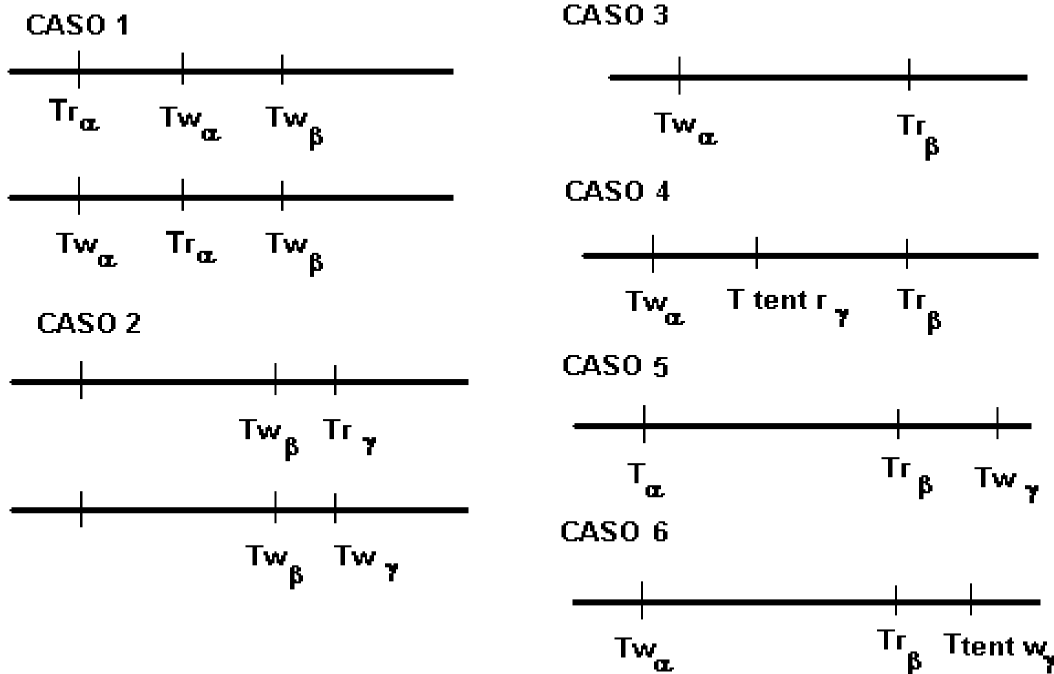
Se utiliza el algoritmo de Lamport para asegurar que esta hora es única.

Cada archivo tiene asociado un sello de lectura y uno de escritura que indican qué TR cumplida (que hizo COMMIT) es la última que leyó o actualizó.

Las propiedades de los sellos temporales respecto del bloqueo son que cuando la TR encuentra un sello mayor aborta en tanto que en el bloqueo puede esperar o continuar. Los sellos temporales están libres de deadlock.

Esta técnica asegura un ordenamiento correcto en el tiempo. Cuando el orden es incorrecto significa que una TR que empezó después que la actual manejó el archivo e hizo COMMIT con lo cual la TR actual es tardía y debe ser abortada.

Veamos algunos ejemplos, para ello asumamos tres transacción: α, β y γ, de las cuales α es la transacción vieja que accedió al recurso y ya hizo COMMIT. Las transacciones α y β se inician concurrentemente y el sello temporal de β es < que el sello temporal de γ.



Veamos primero el caso de la grabación.

En los casos 1) se acepta porque $Tw\beta > Tw\alpha$ y se registra $Tw\beta$ como <u>tentativa</u> de escritura hasta que la transacción realice commit.

En el caso 2) si la transacción γ lee o graba el archivo luego de que la transacción β lo accedió con intención de grabar y la transacción γ hace commit, entonces la transacción β debe ser cancelada.

Veamos ahora ejemplos respecto de la lectura.

En el caso 3) no hay conflicto por lo tanto la lectura procede.

En el caso 4) como la transacción γ intenta leer en una marca temporal menor que el intento de lectura de β, entonces β debe esperar a que γ haga commit y luego leerá.

En el caso 5) la transacción γ modificó el archivo e hizo commit, por lo tanto β debe ser cancelada.

En el caso 6) la transacción γ modifica tentativamente el archivo y si bien aún no hizo commit, β está atrasada y también debe ser cancelada.

La técnica de sellos temporales está libre de Deadlock aunque su implementación es bastante compleja.

23.5. - **ABRAZO MORTAL EN SISTEMAS DISTRIBUIDOS**

El abrazo mortal en sistemas distribuidos es peor, más difícil de evitar, prevenir e incluso de detectar.

Es también difícil de arreglar por que la información está desparramada entre distintas máquinas.

Alguno autores distinguen entre dos tipos de abrazos mortales en sistemas distribuidos, a saber :

- *deadlock de comunicaciones*

Estos son del estilo de : A manda mensaje a B
B manda mensaje a C
C manda mensaje a A

En ciertas circunstancias esto puede llevar a un deadlock.

- *deadlock de recursos*

Cuando dos procesos pelean por el acceso exclusivo a dispositivos de E/S, archivos, bloqueos u otros recursos.

No vamos a usar esta distinción ya que los recursos de comunicación son también recursos e igual se pueden modelizar.

Además las esperas circulares como la descripta no son frecuentes, por ejemplo en el modelo cliente-servidor es raro que el servidor desee enviar un mensaje al cliente que lo llamó actuando él como un cliente y esperando que el cliente actúe como servidor con lo cual la espera circular es extraño que ocurra como consecuencia de solamente la comunicación en sí.

Se utilizan cuatro estrategias para manejar abrazos mortales :

1) - El algoritmo del avestruz (ignorar el problema)
2) - Detección (dejar que ocurra, detectarlo y tratar de recuperar)
3) - Prevención (hacer que estáticamente el abrazo mortal sea imposible desde el punto de vista de la estructura)
4) - Evitar (evitar el abrazo alocando recursos cuidadosamente)

El algoritmo 1) es tan usado como en los sistemas monoprocesadores. En instalaciones de oficina, programación, control de procesos, y muchas otras no existe ningún algoritmo de prevención. No obstante ello aplicaciones de bases de datos pueden implementar su propio algoritmo si lo necesitan.

El algoritmo 2) es muy popular porque los otros dos son muy difíciles.

La prevención (3) es también posible pero aún más difícil de implementar que con un solo procesador. Sin embargo en presencia de las transacciones atómicas algunas nuevas opciones están disponibles y que no las teníamos antes.

El último algoritmo (4) no se usa nunca en sistemas distribuidos. Si no se usa casi en sistemas monoprocesadores ¿porqué usarlo aquí en donde es aún más dificultoso ? El problema es que algoritmos como el del Banquero necesitan conocer de antemano cuantas instancias de los recursos se utilizarán y esta información no está disponible en los sistemas distribuidos.

Veremos solo dos de las técnicas : detección y prevención.

23.5.1. - Detección de abrazo mortal

Cuando se detecta un abrazo mortal en un sistema monoprocesador la solución es matar uno o más procesos lo que deriva en uno o más usuarios infelices. Pero en un sistema distribuido gracias a las propiedades y por el diseño de las transacciones atómicas el matar una o más transacciones no acarrea mayor dificultad. Luego las consecuencias son menos severas al matar un proceso si se usan las transacciones atómicas que si no se usan.

23.5.1.1. - Detección centralizada de Abrazo Mortal

Esto es parecido al caso monoprocesador. Si bien cada máquina tiene los grafos de alocación de sus propios procesos y recursos existe un coordinador centralizado que mantiene el grafo de recursos de todo el sistema.

Cuando el coordinador detecta un ciclo mata un proceso para romper el abrazo.

Este grafo debe actualizarse constantemente.

Una forma es que cuando un arco de agrega o elimina se debe enviar un mensaje al coordinador.

Otra es que cada x tiempo cada proceso envía una lista de arcos a agregar o eliminar (requiere menos mensajes que el anterior).

Otra es que el coordinador pida información cuando la necesita.

Lamentablemente ninguno de estos métodos trabaja bien.

Supongamos la situación (Fig. 23-14) en la máquina 0 en la cual el proceso A tiene el recurso S y espera el recurso R que está asignado el proceso B.

En la máquina 1 el proceso C tiene T y espera por S.

La situación vista por el coordinador es correcta. Tan pronto como B termine a podrá obtener R y terminar liberando S para C.

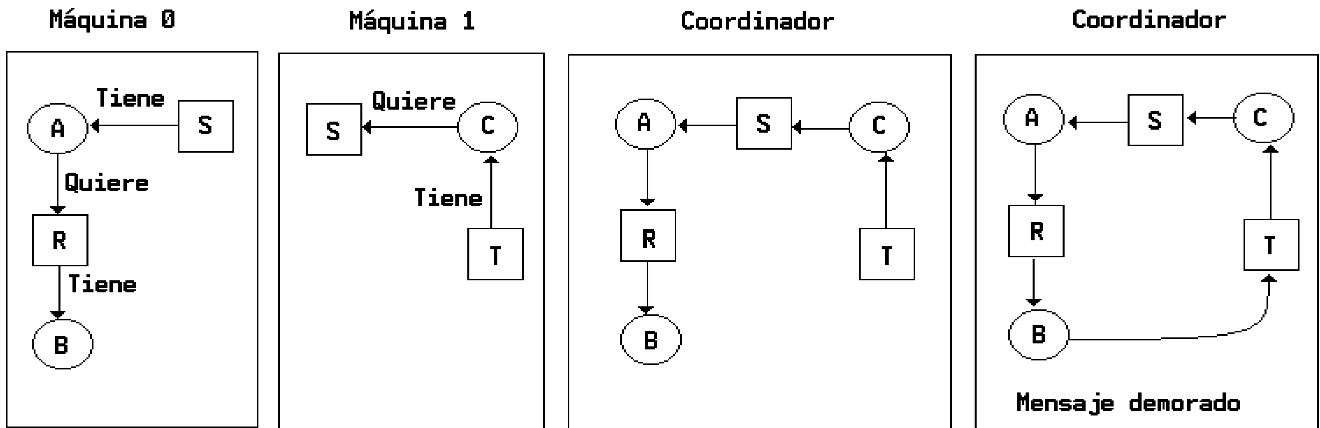


Fig.23.14. - Detección centralizada de Abrazo Mortal.

Luego de un tiempo B libera R y solicita T, y la máquina 1 avisa que B está esperando por su recurso T. Pero el mensaje de la máquina 1 llega primero al coordinador el cual al agregar el arco detecta la existencia de un abrazo mortal llegando a la conclusión de que algún proceso debe morir.

Esta situación es llamada un **falso abrazo mortal** o falso ciclo. Muchos algoritmos en sistemas distribuidos producen falsos abrazos mortales debido a información incompleta o demorada.

Una forma de solución es utilizar el algoritmo de Lamport para proveer un tiempo global. El mensaje de la máquina 1 se dispara luego del requerimiento de la máquina 0 por lo tanto deberá tener un sello temporal mayor.

Cuando el coordinador recibe el mensaje de la máquina 1 que le lleva a sospechar la existencia de un abrazo mortal deberá enviar a todas las máquinas un mensaje que diga : "Acabo de recibir un mensaje con sello temporal T que me provoca abrazo mortal. Alguien tiene un mensaje para mí con menor sello temporal ? Por favor, envíenmelo inmediatamente".

A pesar de que este método elimina los falsos abrazos mortales requiere de tiempo global y es caro. Más aún, existen otras situaciones en las que la eliminación del falso abrazo mortal es aún más difícil.

Puede suceder también que existan rollbacks innecesarios porque mientras el coordinador envió un rollback por alguna razón otro de los procesos envueltos en el deadlock dejó de ejecutar.

23.5.1.2. - Detección jerárquica de abrazo mortal

En este esquema en los nodos superiores se va guardando la información referente a los nodos inferiores, cada uno de los cuales guarda su propio grafo. El ejemplo puede visualizarse en la Fig. 23.15.

23.5.1.3. - Detección distribuida de abrazo mortal

Veamos el algoritmo de Chandy-Misra-Haas (1983), en este algoritmo los procesos pueden pedir múltiples recursos de a una vez en lugar de a uno por vez.

Esto permite acelerar considerablemente la fase de crecimiento de una transacción. La consecuencia es que un proceso puede ahora esperar por do o más recursos simultáneamente.

En el dibujo (Fig. 23-16) eliminamos las flechas de los recursos e indicamos solo el encadenamiento entre los procesos. El proceso 3 en la máquina 1 está esperando dos recursos, uno retenido por el proceso 4 y el otro por el proceso 5. Algunos procesos están esperando recursos locales (proceso 1) y otros esperan recursos de otras máquinas (proceso 2).

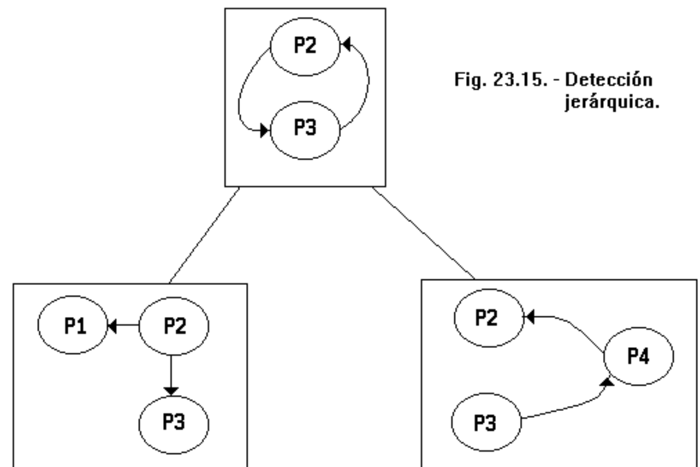


Fig. 23.15. - Detección jerárquica.

Estos últimos arcos son los que hacen dificultoso ver los ciclos.

Se invoca al algoritmo cuando algún proceso tiene que esperar por algún recurso (por ej. el proceso 0 espera por el proceso 1).

En este punto se envía un mensaje de **prueba** a los procesos que retienen el recurso. Este mensaje consta de tres números: el proceso que se está bloqueando, el proceso que envía el mensaje y el proceso que retiene el recurso, en este caso el mensaje contiene la terna (0, 0, 1).

Cuando el mensaje llega el receptor chequea para ver si él mismo está esperando a algún proceso. De ser así actualiza el mensaje manteniendo el primer campo pero reemplazando el segundo por sí mismo y el tercero por el número del proceso al cual está esperando. El mensaje es luego ruteado al proceso sobre el cual él se encuentra bloqueado. Si está bloqueado sobre varios a todos ellos se les envía mensaje (diferente).

El algoritmo es igual aunque el recurso sea local o remoto.

En la figura podemos ver los mensajes (0,2,3), (0,4,6), (0,5,7) y (0,8,0).

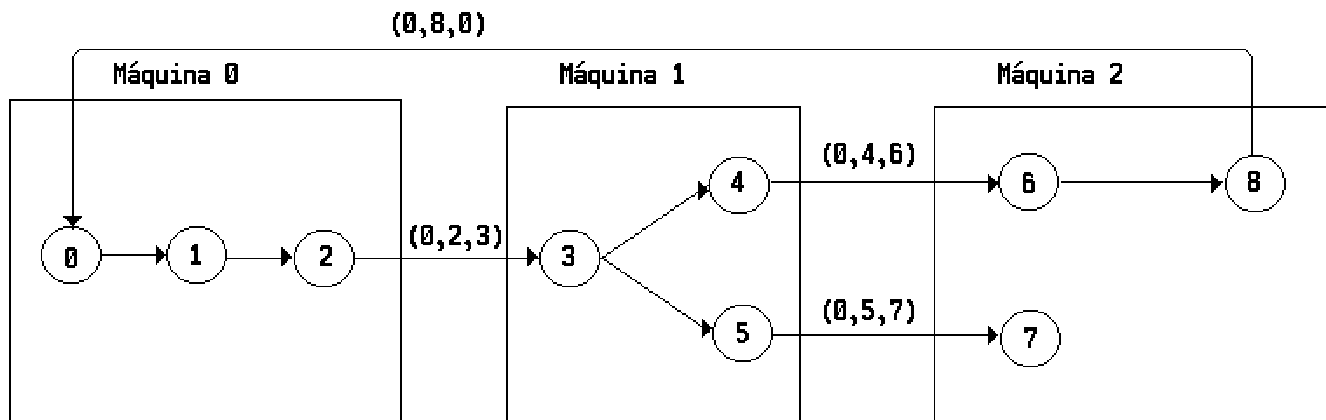


Fig. 23.16. - Detección distribuida.

Si el mensaje da toda la vuelta y vuelve al emisor original entonces existe un abrazo mortal.

Este abrazo se puede romper de varias formas. Por ejemplo el primer emisor del mensaje puede cometer suicidio.

No obstante este método tiene problemas si varios procesos invocan simultáneamente al algoritmo. Supongamos que tanto el proceso 0 como el proceso 6 se bloquean al mismo tiempo y ambos inician mensajes de prueba.

Cada uno descubrirá el abrazo mortal y se suicidará. Esto provoca que mueran demasiados procesos cuando la muerte de uno de ellos bastaba.

Una alternativa sugiere agregar la propia identidad de cada proceso al final del mensaje de manera tal que cuando vuelve al emisor inicial podría listarse el ciclo completo. El emisor podría entonces ver cual proceso tiene el mayor número y enviarle un mensaje solicitándole que se mate. De esta forma varios procesos que descubren el ciclo elegirían la misma víctima.

En el campo de los sistemas distribuidos la teoría difiere aún más de la realidad. Por ejemplo enviar un mensaje de prueba cuando se está bloqueado es algo no trivial.

Existe mucha investigación en este campo pero una vez que hay un método nuevo alguien muestra un contraejemplo.

En conclusión, tenemos mucha investigación, modelos que no se ajustan bien a la realidad, las soluciones halladas son generalmente impracticables, los análisis de performance dados son poco significativos y los resultados probados son a menudo incorrectos. Como algo positivo esta es un área que ofrece grandes oportunidades de mejoras.

23.5.2. Prevención distribuida de abrazo mortal

La prevención consiste en diseñar cuidadosamente el sistema de manera tal que los deadlocks sea estructuralmente imposibles.

Muchas técnicas incluyen :

- los procesos sólo pueden retener de a un recurso por vez,
- los procesos deben requerir todos sus recursos al inicio,
- los procesos deben liberar todos sus recursos al requerir uno nuevo.

Estas prácticas son engorrosas en la práctica.

Otro método es el ordenamiento de los recursos y exigir que los requerimientos de los procesos se realicen estrictamente en ese orden (vimos que está libre de deadlocks).

No obstante en un sistema distribuido con tiempos globales y transacciones atómicas existen otros dos métodos que son posibles.

Ambos se basan en asignarle a cada transacción un sello temporal en el momento en que la misma se inicia. Es esencial que dos transacciones no puedan tener nunca el mismo sello temporal para ello se puede utilizar el algoritmo de Lamport.

La idea es que cuando un proceso se va a bloquear en espera de un recurso que está utilizando otro proceso se chequea cuál de los dos tiene un mayor sello temporal (cuál es el más joven). Se puede permitir entonces el bloqueo si el que se bloquea tiene un menor sello temporal, es decir es más viejo que el que posee el recurso en ese momento.

De esta forma siguiendo una cadena de procesos en espera el sello temporal de ellos siempre va in crescendo con lo cual la creación de ciclos es imposible.

También es factible permitir el bloqueo si el que se quiere bloquear es más joven que el que retiene el recurso (es más joven) en cuyo caso la cadena de procesos en espera tiene sellos temporales que van decreciendo.

Si bien ambos métodos previenen el abrazo mortal es sensato otorgar prioridad a los procesos más viejos ya que han procesado por más tiempo, el sistema invirtió mucho ya en ellos y es probable que retengan más recursos. Además un proceso joven que se suicida eventualmente envejece al punto tal en que es el más viejo y de esta forma se elimina la inanición.

Como vimos antes matar una transacción es relativamente inocuo ya que por definición se la puede restaurar en forma segura más tarde.

Para aclarar consideremos la situación de la figura 23-17. En (a) un proceso viejo desea un recurso retenido por uno más joven. En (b) un proceso joven desea un recurso retenido por uno más viejo.

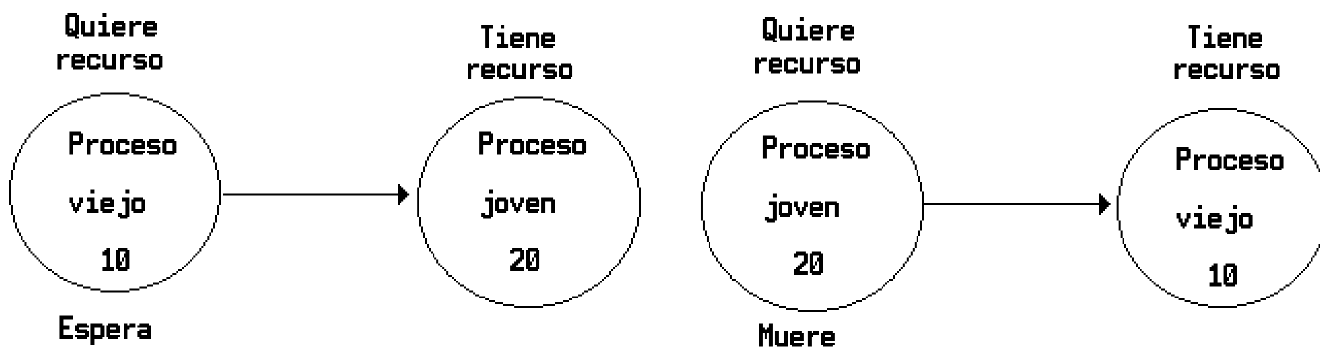


Fig. 23.17. - El algoritmo Wait-Die.

En un caso permitiremos la espera en tanto que en otro matamos al proceso. Supongamos que designamos que en (a) muere y en (B) espera. Lo que estamos haciendo es matar un proceso más viejo que desea un recurso retenido por uno más joven, esto es ineficiente, entonces tenemos que designarlos de otra manera que es como se ve en la figura.

Bajo estas condiciones las flechas apuntan siempre en una dirección creciente de número de transacción haciendo que los ciclos sean imposibles de ocurrir. Este algoritmo se denomina espera-muerte (wait-die).

Una vez que asumimos la existencia de transacciones podemos hacer algo que estaba prohibido anteriormente: alejar los recursos de los procesos que se ejecutan.

Estamos diciendo que cuando existe un conflicto tanto podemos matar al proceso que realiza el requerimiento como al propietario del recurso.

Sin transacciones matar un proceso puede tener consecuencias severas en cuanto a lo que el proceso modificó, etc. Con transacciones estos efectos desaparecen mágicamente cuando la transacción muere.

Consideremos la situación de la figura 23-18 en la cual vamos a permitir el desalojo. Ya que nuestro sistema cree en el trabajo de los ancestros no deseamos que un joven mequetrefe desaloje a un sabio anciano, y por tal motivo rotulamos la parte a) como desalojo y la parte b) como espera.

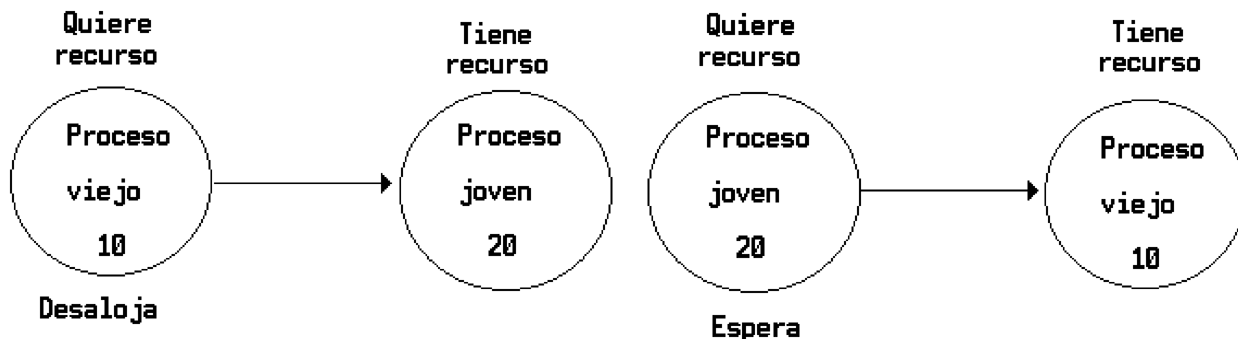


Fig. 23.18. - El algoritmo Wound-Wait.

Este algoritmo se denomina golpear-esperar (wound-wait) debido a que una transacción supuestamente es golpeada o herida (y en realidad muere) y la otra espera.

En el caso a) el pedido de la transacción más vieja provoca la muerte de la transacción más joven desalojándola. El joven comenzará de nuevo y pedirá el recurso otra vez lo que lo colocará en espera como se ve en la parte b).

En el método espera-muere si un proceso viejo quiere un recurso retenido por un joven espera. En cambio si es un joven el que quiere el recurso retenido por un viejo el joven muere, comenzará de nuevo y pedirá el recur-

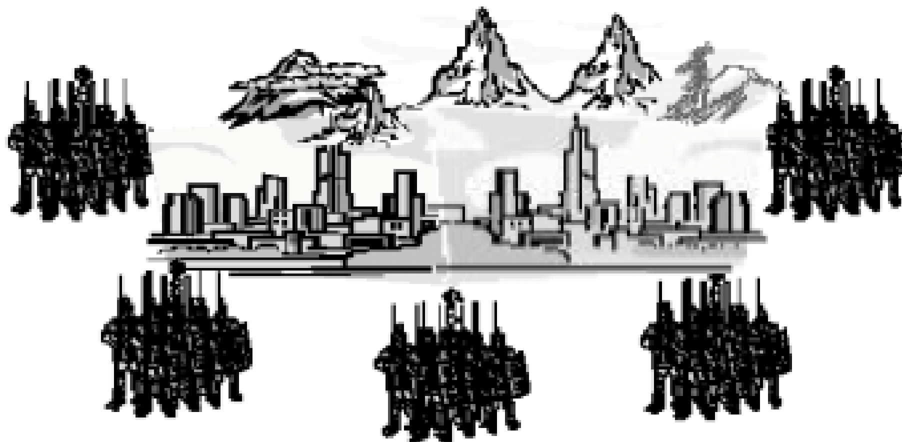
so muriendo nuevamente, y así siguiendo hasta que el proceso viejo libere el recurso. Esta característica desagradable no existe en el método golpear-esperar.

Ambos esquemas permiten la inanición pues las "horas" (que son dadas a cada proceso en el momento de su creación) no son actualizadas.

Para evitar la inanición en ambos esquemas es importante que a una nueva transacción no le asigna un nuevo sello temporal cuando se la reinicia luego de haber sido abortada (una transacción joven de esta forma se volverá más vieja con el transcurso del tiempo y no será cancelada).

23.6. CONSENSO EN PRESENCIA DE FALLAS - El problema de los Generales Bizantinos

Hasta ahora no hemos visto como lograr que un grupo de nodos llegue a un acuerdo considerando la existencia de fallas. Este problema suele mencionarse en la bibliografía como *el problema de los Generales Bizantinos* planteado originalmente por Lamport , Shostak y Peace en una publicación de la ACM en Julio de 1982.



Un grupo de generales sitia una ciudad y deben ponerse de acuerdo en un plan de ataque, ya sea atacar o retirarse, independientemente de que existan generales traidores. Los generales solo se comunican a través de mensajes a los otros generales.

Esta traición puede verse de dos formas:

- los mensajes pueden no llegar, o dicho de otra forma, las comunicaciones son no confiables
- un general traidor puede mentir, o sea, un nodo puede fallar de manera impredecible.

23.6.1. – **Definición del problema**

Un general comandante debe enviar una orden a sus n-1 tenientes generales de manera tal que (Dada una red con n procesos que se comunican entre sí solo a través del pasaje de mensajes sobre canales bidireccionales, asegurar que un proceso envía un ítem a los otros n-1 procesos de manera tal que) :

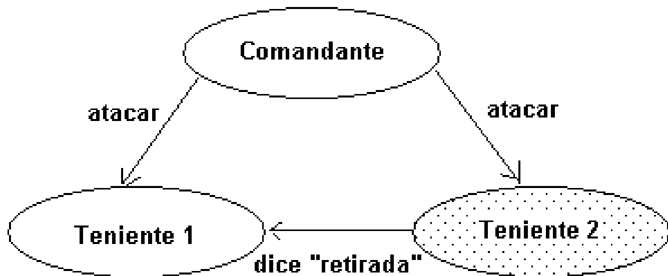
Premisa 1) Todos los tenientes leales obedecen la misma orden (los procesos confiables reciben el mismo ítem)

Premisa 2) Si el general comandante es leal, entonces todos los tenientes leales obedecen la orden que el envió (si el proceso que envía el mensaje es confiable entonces el ítem recibido es igual al ítem enviado)

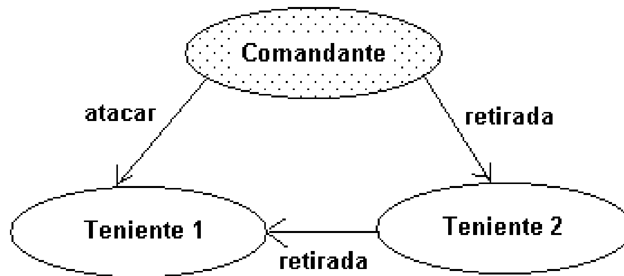
Las premisas 1) y 2) son conocidas como las condiciones de **consistencia interactivas**. Obsérvese que si el comandante es leal la condición P1) se deriva de la condición P2). Sin embargo el comandante puede ser un traidor.

23.6.2. – **Resultados de imposibilidad**

Este problema no es siempre soluble, imagínese el caso de 3 generales :



Caso A - El teniente 2 es un traidor



Caso B - El comandante es un traidor.

En el caso A) para satisfacer P2) el teniente 1 debería atacar. En el caso B) si el teniente 1 ataca viola P1) El teniente 1 no puede distinguir entre ambos casos con la información de que dispone.

No existe solución para el caso de 3 generales que operan en presencia de un traidor.
Generalizando: No existe solución con menos de **3 m + 1** generales cuando existen **m** traidores.

23.6.3. – **Una solución con mensajes sin firma (MSF)**.

Se realizan las siguientes presunciones a priori :

A1)- cada mensaje que se envía se entrega correctamente
A2)- El receptor de un mensaje sabe quién lo envió
A3)- La ausencia de un mensaje puede ser detectada

Las presunciones A1 y A2 evitan que un traidor interfiera las comunicaciones entre dos generales, ya que por A1 no puede interferir con los mensajes que se envían y por A2 no puede provocar confusión introduciendo mensajes espurios. A3 impide al traidor que otro general tome una decisión por la sencilla razón de no haber recibido un mensaje.

En un sistema distribuido las presunciones A1 y A2 indican que en la comunicación la falla del link de conexión se considera como una más de las fallas **m** (un traidor) ya que no se puede distinguir entre una falla de conexión o un proceso que falle en un nodo. La presunción A3 requiere que el emisor y receptor tengan una mecanismo de sincronización de relojes y además que se conozca el tiempo máximo de generación y transmisión de un mensaje.

23.6.3.1. – **El algoritmo MSF**

Este algoritmo vale para **n** generales y **m** traidores con **n > 3 m**.

Se requiere de un valor por defecto $\mathbf{v_{def}}$ por si el general traidor no envía un mensaje (por ejemplo RETIRADA)

Se define una función **mayoría(** $\mathbf{v_1,.....,v_{n-1}}$**) = v** si la mayoría de los valores de $\mathbf{v_j = v}$.

Algoritmo MSF(n, 0) No hay traidores

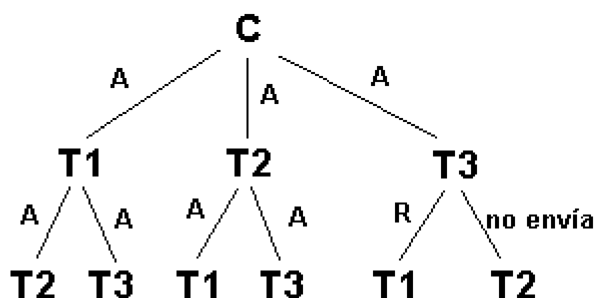
(1) El general comandante envía **v** a cada teniente
(2) Cada teniente utiliza el valor recibido del comandante o $\mathbf{v_{def}}$ si no recibe un valor

Algoritmo MSF(n, m) Hay **m** traidores

(1) El general comandante envía **v** a cada teniente
(2) Para cada **Teniente** $_\mathbf{i}$
 Sea $\mathbf{v_i}$ el valor recibido del comandante o $\mathbf{v_{def}}$ si no recibe un valor
 Enviar $\mathbf{v_i}$ a los n-2 otros tenientes utilizando **MSF(n-1, m-1)**
(3) Para cada **i** y cada **z** con $\mathbf{z \neq i}$
 Sea $\mathbf{v_j}$ = el valor que el $\text{Teniente}_\mathbf{i}$ recibió del $\text{Teniente}_\mathbf{z}$ en el paso (2) o $\mathbf{v_{def}}$ si no recibe un valor
 El $\text{Teniente}_\mathbf{i}$ utiliza el valor de la función **mayoría(** $\mathbf{v_1,.....,v_{n-1}}$**)**

Veamos este algoritmo con un par de ejemplos. Supongamos que el mensaje enviado por el Comandante es ATACAR (o A abreviadamente) y el valor por defecto es RETIRADA (o R abreviadamente).

1er Caso: El Teniente es traidor



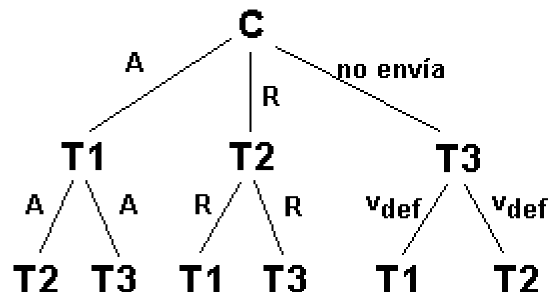
Al final del paso 1 tenemos que:
L1 : v1 = A
L2 : v2 = A
L3 : v3 = A
Al finalizar el paso 3 se tiene que:
L1 : v1 = A v2 = A y v3 = R
L2 : v1 = A v2 = A y v3 = R (valor por defecto)
L3 : v1 = A v2 = A y v3 = A

Al final de esta etapa cada Teniente tiene un conjunto de valores y arriba a la misma decisión (respeta P1) y el valor enviado por C es el valor de la función **mayoría** (respeta P2).

2do Caso: El Comandante es traidor

Al final del paso 1 tenemos que:

L1 : v1 = A
L2 : v2 = R
L3 : v3 = R (valor por defecto)
Al finalizar el paso 3 se tiene que:
L1 : v1 = A v2 = R y v3 = R (valor por defecto)
L2 : v1 = A v2 = R y v3 = R (valor por defecto)
L3 : v1 = A v2 = R y v3 = R (valor por defecto)



Los tres tenientes leales reciben el mismo valor de la función mayoría y las premisas P1 y P2 se respetan. Nótese en este caso que como el Comandante **no** es leal su orden no es acatada aunque todos los tenientes actúan coordinadamente.

Un lema que se desprende de este algoritmo indica que : Para cualquier **m** y **k** el algoritmo **MSF(m)** satisface la Premisa 2 si hay al menos **2k+m** generales y a lo sumo **m** traidores.

Por otra parte para cualquier **m** el algoritmo **MSF(m)** satisface las premisas 1 y 2 si hay más de **3m** generales y a lo sumo **m** traidores.

26.6.3.1. – **Complejidad de MSF(n,m)**

Al aplicar **MSF(n,m)** primero se envía **n-1** mensajes. Cada mensaje invoca **MSF(n-1, m-1)** lo que provoca que se envíen **n-2** mensajes, etc.

Etapa 1 : **(n-1)** mensajes
Etapa 2 : **(n-1)(n-2)** mensajes
....
Etapa m+1 : **(n-1)(n-2).....(n-(m+1))** mensajes

Luego, el total de mensajes es **$O(n^{m+1})$**

Nótese que las **m+1** etapas de intercambio de mensajes es una característica fundamental de los algoritmos en los que se logra consenso en presencia de **m** posibles procesos que fallen.

23.6.4. – **Una solución con mensajes firmados MF**

Lo que se busca es restringir la posibilidad de que el traidor mienta permitiendo a los generales enviar mensajes firmados que no pueden ser alterados.

Se agrega una presunción a las hechas anteriormente:

A4): (a) La firma de un general leal no puede ser alterada y cualquier alteración del contenido de sus mensajes firmados puede ser detectado.
(b) Cualquiera puede verificar la autenticidad de la firma del general

Adoptaremos la siguiente notación:

V:j:i – valor **v** firmado por **j** y luego el valor **v:j** ha sido firmado por **i**. El general 0 es el comandante.

Se necesita una función **elección(v)** la cual selecciona el valor de **v** de un conjunto de valores de **v** de forma tal que :

Elección ({v}) = v;
Elección ({v_{def}}) = v_{def};

Esta función es utilizada para obtener el valor del consenso y no necesariamente es el valor de la mayoría ni un promedio.

23.6.4.1. – **El algoritmo MF**

En este algoritmo cada teniente **i** mantiene un conjunto **V_i** de ordenes correctas firmadas que ya ha recibido. Con un comandante leal el conjunto no contiene más de un elemento.

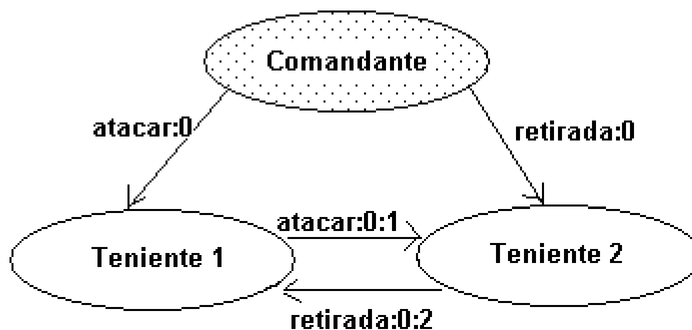
Algoritmo MF(m) Inicialmente **V_i = {}**

(1) El comandante envía su valor firmado a cada teniente
(2) Para cada **i** :
  (a) Si el teniente **i** recibe un mensaje **v:0** y aún no ha recibido una orden, entonces:
    (i) establece **V_i = {v}**
    (ii) envía **v:0:i** a los otros tenientes
  (b) Si el teniente **i** recibe un mensaje de la forma **v:0:j_1: ... :j_k** y **v** no está en el conjunto **V_i** entonces:

(i) $V_i := V_i + \{v\}$
(ii) Si **k < m** entonces envía el mensaje $v{:}0{:}j_1{:} \ldots {:}j_k{:}i$ a cada teniente de $j_1{:} \ldots {:}j_k$

(3) Cuando no se reciben más mensajes el teniente **i** obedece la orden que surge de **elección(V_i)**

Veamos un ejemplo:



v1 = v2 = elección({atacar,retirada})

Nótese que con mensajes firmados los tenientes pueden detectar que el comandante es un traidor debido a que su firma aparece en dos órdenes opuestas y por el supuesto A4 solo él pudo haberlas firmado.

En este caso siempre es posible lograr un acuerdo en tanto y en cuanto existan por lo menos dos generales leales.

26.3.4.2.- **Complejidad**

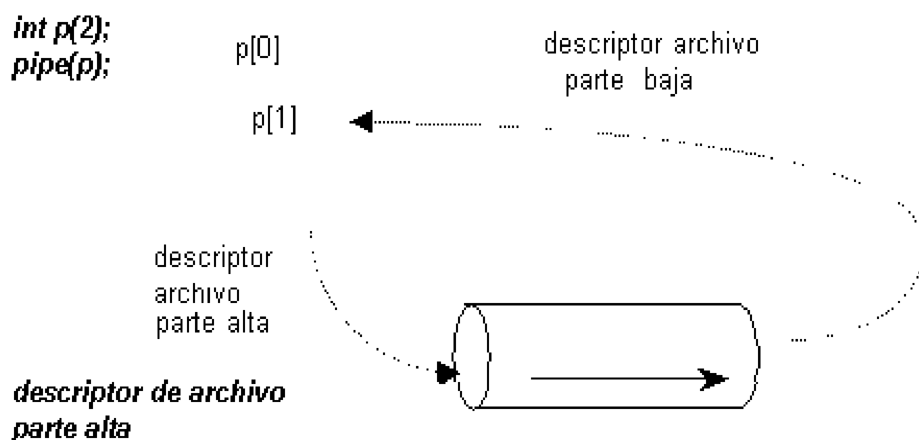
La cantidad de mensajes es del orden de $O(n^{m+1})$
La cantidad de etapas : **m + 1**

23.7. COMUNICACIÓN ENTRE PROCESOS - INTER PROCESS COMMUNICATION (IPC)

Hasta ahora hemos visto mecanismos de comunicación entre procesos con uso de memoria común, como semáforos, regiones críticas, monitores, etc. y en forma incipiente, mecanismos que no utilizan memoria común, por medio de primitivas send y receive, el modelo cliente/servidor y una introducción al RPC.

A continuación, haremos una breve introducción a un mecanismo de comunicación que utiliza archivos especiales, los pipes, para luego extender el concepto a mecanismos de comunicación que utilizan un principio similar, los sockets, pero que permiten la comunicación de procesos que residen en nodos (computadoras) ligados por algún medio de comunicación.

23.7.1. PIPES



Los pipes o tubos son archivos que permiten un mecanismo de comunicación entre procesos que cuenta de dos partes:

Alta: por la cual es posible escribir

Baja: desde la cual se puede leer

Se trata de un mecanismo que maneja una estructura de cola (FIFO) en el cual la información transmitida por un lado es recibida y puede ser retirada por el otro lado.

La comunicación es unidireccional, sincrónica y unioperacional, pues no es posible escribir y leer al mismo tiempo. O sea que mientras un proceso realiza una de las dos operaciones permitidas (en realidad la que se le permite) el otro espera su finalización para comenzar la suya.

Al estar compartiendo un mismo recurso, el pipe, este debe ser heredado de otro proceso que lo haya generado, o sea son utilizados para establecer comunicación entre dos procesos que posean el mismo padre.

Se crean a partir de una llamada al sistema pipe() (ver figura)

Los pipes tienen una capacidad finita.

Si el proceso de la parte alta del pipe escribe en él y el pipe está lleno, el proceso se bloqueará hasta que la parte baja libere datos.

Si el proceso de la parte baja del pipe realiza una lectura y el pipe está vacío el proceso se bloquea.

Por ejemplo si se tiene:

While ((count = read(fd, buffer, 100)) > 0 { procesa }

Si suponemos que el buffer tiene 230 bytes, luego de dos lecturas, quedan 30, que será lo que leerá en la tercera, en su cuarto intento se bloqueará.

Si un proceso intenta leer la parte baja del pipe, que no tiene abierta su parte alta, la lectura regresará un EOF (fin de archivo).

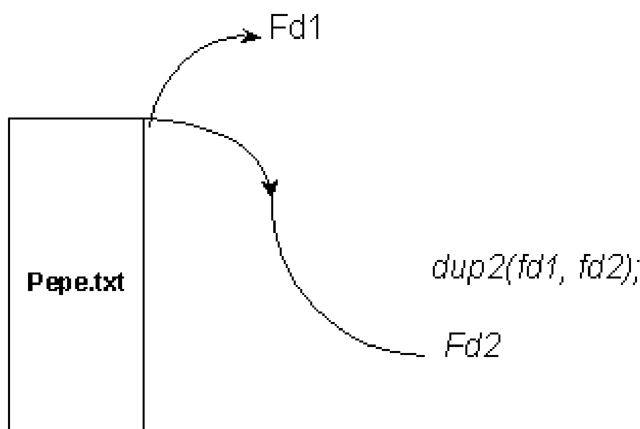
Como los pipes son archivos, es necesario generar sus descriptores, en general se reutiliza un archivo ya abierto por el padre, en cuyo caso se utiliza una llamada al sistema que duplica un descriptor de un archivo ya abierto (**dup2()**), de la manera que se aprecia en la figura.

#include <unistd.h>

int dup2(int fd1, int fd2);

Cuyo resultado es:

- El descriptor de archivos fd2 va a apuntar al mismo archivo que fd1.
- Fd1 es un descriptor de archivos que hace referencia a un archivo abierto.
- Fd2 es un entero no negativo menor al número de descriptores de archivos abiertos permitidos.
- Si fd2 apuntaba a un archivo abierto, diferente de fd1, primero cierra el archivo en cuestión y luego apunta al de fd1.



23.7.2. **Ejemplos de uso de pipes**

Veamos un sencillo ejemplo en el que se utilizan dos comandos UNIX, **who I more**

```
/* Archivo del programa whomore.c  */
main()
{
     int fds[2]

     pipe(fds);
     /* Hijo1 reconecta stdin a  parte baja del pipe y cierra alta */
     if (fork() == 0)  {
       dup2(fds[0], 0);
       close(fds[1]);
       execlp("more","more",0);
    }
    else {
      /* Hijo2 reconecta stdout  a  parte alta del pipe y cierra baja */
       if ( fork() == 0 )      {
           dup2(fds[1], 1);
           close(fds[0]);
           execlp("who", "who", 0);
       }
       else {        /* padre cierra ambas partes y espera a los hijos */
           close(fds[0]);
           close(fds[1]);
           wait(0);
           wait(0):
       }
}
```

Veamos otro ejemplo en que dos "hermanos" mantienen comunicación por **pipes**

```
#include <stdio.h>
#define BLKSIZE 80

main()
   int n;
   char msg[80];
   int fd[2];

   if  ( pipe(fd) < 0 ) {
      fprintf(stderr,"Error en la creación del pipe \n");
      exit(1);
   }
   if ( fork() == 0) {
       close(fd[0]);
       sprintf(msg, "Mensaje envía por %d",getpid());
       write(fd[1], msg, BLKSIZE)
   }
   else
       if ( fork() == 0) {
          close(fd[1]);
          read(fd[0], msg, BLKSIZE);
          printf("Proceso %d recibió mensaje: %s \n",getpid(), msg);
       }
       else {
          close(fd[0]);  close(fd[1]);
          wait(0);   wait(0);
          printf("Los hijos terminaron de comunicarse \n");
       }
}
```

Las limitaciones de los pipes residen en:

1. Pipes son unidireccionales. La solución para lograr comunicación en dos sentidos es crear dos pipes.
2. Pipes no pueden autentificar al proceso con el que mantiene comunicación.
3. Pipes deben de ser pre-arreglados. Dos procesos no relacionados no pueden conectarse vía pipes, deben tener un ancestro común que cree el pipe y se los herede.
4. Pipes no trabajan a través de una red. Los dos procesos deben encontrarse en la misma máquina.

23.8. **SOCKETS**

Los sockets no son mas que puntos o mecanismos de comunicación entre procesos que permiten que un proceso hable (emita o reciba información) con otro proceso incluso estando estos procesos en distintas máquinas. Esta característica de interconectividad entre máquinas hace que el concepto de socket nos sirva de gran utilidad. Esta interfaz de comunicaciones es una de las contribuciones Berkeley al sistema UNIX, implementándose las utilidades de interconectividad de este Sistema Operativo (rlogin, telnet, ftp,...) usando sockets.

La forma de referenciar un socket por los procesos implicados es mediante un descriptor del mismo tipo que el utilizado para referenciar archivos. De hecho muchos desarrolladores dicen que un socket no es más que un pipe con un protocolo asociado y bidireccional.
Debido a esta característica, se podrá realizar redirecciones de los archivos de E/S estándar (descriptores 0,1 y 2) a los sockets y así combinar entre ellos aplicaciones de la red. Todo nuevo proceso creado heredará, por tanto, los descriptores de sockets de su padre.

La comunicación entre procesos a través de sockets se basa en la filosofía CLIENTE/SERVIDOR: un proceso en esta comunicación actuará de proceso servidor creando un socket cuyo nombre conocerá el proceso cliente, el cual podrá "hablar" con el proceso servidor a través de la conexión con dicho socket nombrado.

El proceso crea un socket sin nombre cuyo valor de retorno es un descriptor sobre el que se leerá o escribirá, permitiéndose una comunicación bidireccional, característica propia de los sockets y que los diferencia de los pipes, o canales de comunicación unidireccional entre procesos de una misma máquina. El mecanismo de comunicación vía sockets tiene los siguientes pasos:

1º) El proceso servidor crea un socket con nombre y espera la conexión.
2º) El proceso cliente crea un socket sin nombre.
3º) El proceso cliente realiza una petición de conexión al socket servidor.
4º) El cliente realiza la conexión a través de su socket mientras el proceso servidor mantiene el socket servidor original con nombre.

Es muy común en este tipo de comunicación lanzar un proceso hijo, una vez realizada la conexión, que se ocupe del intercambio de información con el proceso cliente mientras el proceso padre servidor sigue aceptando conexiones. Para eliminar esta característica se cerrará el descriptor del socket servidor con nombre en cuanto realice una conexión con un proceso socket cliente.

Antes de seguir avanzando repasemos nuestros conocimientos sobre redes y en particular sobre el modelo TCP/IP

23.8.1. **El modelo TCP/IP**.

Los cinco niveles del modelo TCP/IP son:

	TCP/IP	OSI
(NFS)		7. Aplicación
(XDR)	5. Aplicación	6. Presentación
(RPC)		5. Sesión
(TCP/UDP)	4. Transporte	4. Transporte
(IP/ICMP)	3. Internet	3. Red
Trama Ethernet	2. Interfaz de Red	2. Enlace de Datos
Red Ethernet	1.Hardware	1. Físico

Nivel Interfaz de Red y Hardware:

Agrupa los bits en tramas para el manejo de la información y se ocupa de las características técnicas de la red (voltajes, pines, ...)

Nivel Internet:

Se corresponde con el nivel de Red OSI y controla el direccionamiento de la información. El IP se trata de un protocolo para el intercambio de datagramas en modo no conectado. Esto no garantiza la llegada de mensajes (cosa que se hará con el TCP). El algoritmo de direccionamiento de Internet se basa en tablas de direccionamiento de los datagramas difundidos por los gateways.

Nivel de Transporte:

Se corresponde con el de Transporte OSI, garantizando la seguridad de la conexión y el control del flujo. Incluye el Protocolo de Control de Transmisión (TCP) y el Protocolo de Datagrama de Usuario (UDP).

* El TCP es un protocolo orientado a conexión que transporta de forma segura grupos de octetos (segmentos) modo dúplex (en los dos sentidos).
* Utiliza el mecanismo de puerto (al igual que el protocolo de transporte UDP, pero que actúa en modo datagrama no conectado).

Este mecanismo se basa en la asignación para cada uno de los protocolos del nivel de transporte (TCP o UDP) de un conjunto de puertos de E/S identificados mediante un número entero. Así TCP y UDP multiplexarán las conexiones por medio de los números de los puertos. Existen una serie de puertos reservados a aplicaciones estándares Internet (ECHO - puerto 7, FTP - puerto 21, TELNET - puerto 23). El archivo **/etc/services** contiene la lista de los puertos estándar. En UNIX están reservados los números de puerto inferiores a 1024. El resto pueden ser utilizados, cualidad fundamental que aprovechan los programas definidos por el usuario para el establecimiento de comunicaciones entre nodos.

Nivel de Aplicación:

Incluye los niveles OSI de sesión, presentación y aplicación. Ejemplos de estos niveles son el telnet, ftp o el sistema de archivos de red NFS para el nivel de aplicación, el lenguaje de descripción de información XDR para el nivel de presentación o la interfaz de llamada a procedimientos remotos RPC.

Por ejemplo, el formato de una trama telnet sería con sus archivos UNIX asociados:

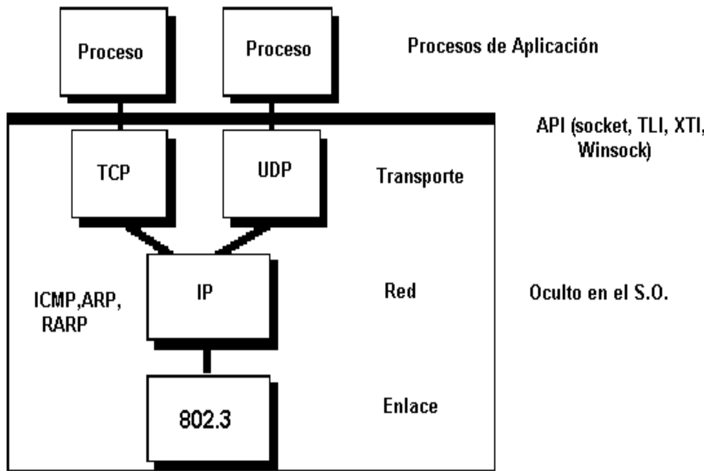
/etc/services

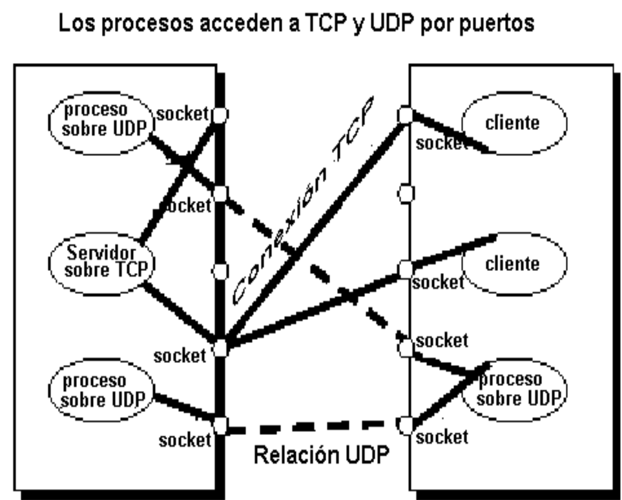
Dirección Ethernet	IP	TCP	telnetd

/etc/host /etc/protocols inetd.conf

En forma gráfica sería como puede visualizarse en la figura.

23.8.2. **Primera aproximación a los sockets**





Todo socket viene definido por dos características fundamentales:

- El **tipo del socket**, que indica la naturaleza del mismo, el tipo de comunicación que puede generarse entre los sockets.
- El **dominio del socke**t especifica el conjunto de sockets que pueden establecer una comunicación con el mismo.

Vamos a estudiar con más detalle estos dos aspectos:

23.8.2.1. **Tipos de sockets.**

Define las propiedades de las comunicaciones en las que se ve envuelto un socket, esto es, el tipo de comunicación que se puede dar entre cliente y servidor. Estas pueden ser:

- Fiabilidad de transmisión.
- Mantenimiento del orden de los datos.
- No duplicación de los datos.
- El "Modo Conectado" en la comunicación.
- Envío de mensajes urgentes.

Los tipos disponibles son los siguientes:

* Tipo SOCK_DGRAM: sockets para comunicaciones en modo no conectado, con envío de datagramas de tamaño limitado (tipo telegrama).
En dominios Internet como la que nos ocupa el protocolo del nivel de transporte sobre el que se basa es el UDP.

* Tipo SOCK_STREAM: para comunicaciones confiables en modo conectado, de dos vías y con tamaño variable de los mensajes de datos. Por debajo, en dominios Internet, subyace el protocolo TCP.

* Tipo SOCK_RAW: permite el acceso a protocolos de más bajo nivel como el IP (nivel de red)

* Tipo SOCK_SEQPACKET: tiene las características del SOCK_STREAM pero además el tamaño de los mensajes es fijo.

23.8.2.2. **El dominio de un socket.**

Indica el formato de las direcciones que podrán tomar los sockets y los protocolos que soportarán dichos sockets.

La estructura genérica es

```
struct sockaddr {
        u__short    sa__family;        /* familia */
        char        sa__data[14];      /* dirección */
    };
```

Pueden ser:

* Dominio AF_UNIX (Address Family UNIX):

El cliente y el servidor deben estar en la misma máquina. Debe incluirse el archivo cabecera /usr/include/sys/un.h. La estructura de una dirección en este dominio es:

```
struct sockaddr__un {
short        sun__family;  /* en este caso AF_UNIX */
char         sun__data[108]; /* dirección */
    };
```

* Dominio AF_INET (Address Family INET):

El cliente y el servidor pueden estar en cualquier máquina de la red Internet. Deben incluirse los archivos cabecera /usr/include/netinet/in.h, /usr/include/arpa/inet.h, /usr/include/netdb.h. La estructura de una dirección en este dominio es:

```
struct in__addr {
    u__long    s__addr;
};

struct sockaddr__in {
short        sin_family;  /* en este caso AF_INET */
u__short     sin_port;   /* numero del puerto */
struct in__addr         sin__addr; /* direcc Internet */
char         sin_zero[8];    /* campo de 8 ceros */
    };
```

Estos dominios van a ser los utilizados en xshine. Pero existen otros como:
* Dominio AF_NS:
Servidor y cliente deben estar en una red XEROX.
* Dominio AF_CCITT:
Para protocolos CCITT, protocolos X25, ...

23.8.3. **FILOSOFIA CLIENTE-SERVIDOR:**

23.8.3.1. **El Servidor.**

Vamos a explicar el proceso de comunicación servidor-cliente en modo conectado, modo utilizado por las aplicaciones estándar de Internet (telnet, ftp). El servidor es el proceso que crea el socket no nombrado y acepta las conexiones a él. El orden de las llamadas al sistema para la realización de esta función es:

1º) int socket (int dominio, int tipo, int protocolo)
crea un socket sin nombre de un dominio, tipo y protocolo específico
dominio : AF_INET, AF_UNIX
tipo : SOCK__DGRAM, SOCK__STREAM
protocolo : 0 (protocolo por defecto)

2º) int bind (int dfServer, struct sockaddr* direccServer, int longDirecc)

nombra un socket: asocia el socket no nombrado de descriptor dfServer con la dirección del socket almacenado en direccServer. La dirección depende de si estamos en un dominio AF_UNIX o AF_INET.

3º) int listen (int dfServer, int longCola)

especifica el máximo número de peticiones de conexión pendientes.

4º) int accept (int dfServer, struct sockaddr* direccCliente, int* longDireccCli)

escucha al socket nombrado servidor dfServer y espera hasta que se reciba la petición de la conexión de un cliente. Al ocurrir esta incidencia,
crea un socket sin nombre con las mismas características que el socket servidor original, lo conecta al socket cliente y devuelve un descriptor de archivo que puede ser utilizado para la comunicación con el cliente.

23.8.3.2. **El Cliente**

Es el proceso encargado de crear un socket sin nombre y posteriormente enlazarlo con el socket servidor nombrado. O sea, es el proceso que demanda una conexión al servidor. La secuencia de llamadas al sistema es:

1º) int socket (int dominio, int tipo, int protocolo)

crea un socket sin nombre de un dominio, tipo y protocolo específico
dominio : AF_INET, AF_UNIX
tipo : SOCK__DGRAM, SOCK__STREAM
protocolo : 0 (protocolo por defecto)

2º) int connect (int dfCliente, struct sockaddr* direccServer, int longDirecc)

intenta conectar con un socket servidor cuya dirección se encuentra incluida en la estructura apuntada por direccServer. El descriptor dfCliente se utilizará para comunicar con el socket servidor. El tipo de estructura dependerá del dominio en que nos encontremos.

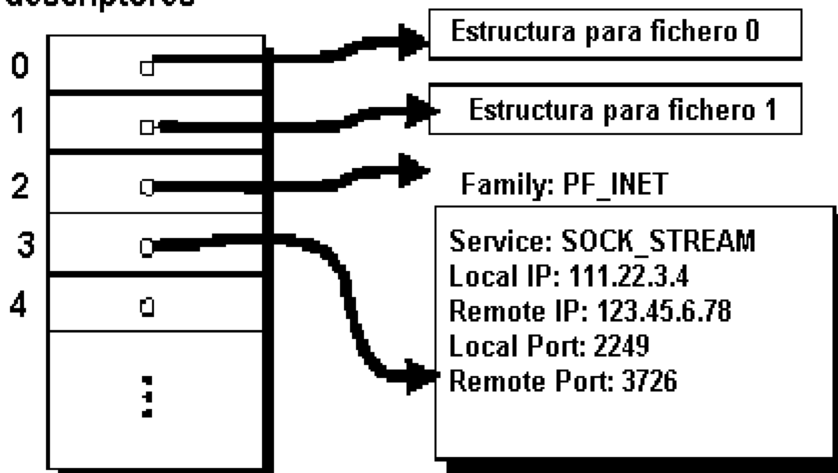
Una vez establecida la comunicación, los descriptores de archivos serán utilizados para almacenar la información a leer o escribir.

En forma esquemática se puede ver así:

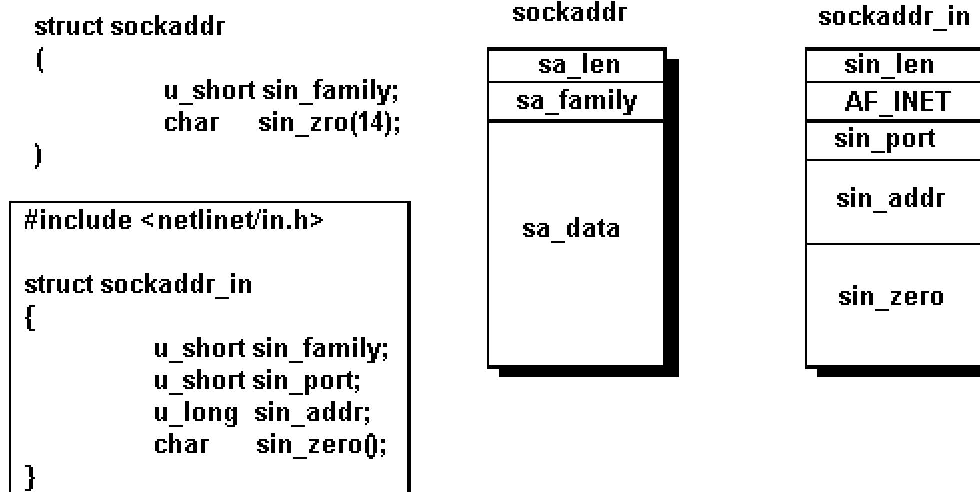
SERVIDOR	CLIENTE
DescrServer = **socket** (dominio, SOCK_STREAM,PROTOCOLO)	descrClient = **socket** (dominio, SOCK_STREAM,PROTOCOLO)
bind (descrServer, PuntSockServer,longServer)	
	do {
listen (descrServer, longCola)	
DescrClient = **accept** (descrServer,PuntSockClient,longClient)	result = **connect** (descrClient, PuntSockServer,longserver)
[close (descrServer)]	} while (result == -1)
< DIALOGO >	< DIALOGO >
close (descrClient)	**close** (descrClient)

23.8.4. **Estructuras de los sockets**

Una descripción gráfica de un socket como descriptor similar a la de archivos



Una descripción gráfica de las estructuras los sockets, (la estructura depende de la familia de protocolos):



Las estructuras son usadas en la programación de sockets para almacenar información sobre direcciones. La primera de ellas es struct sockaddr, la cual contiene información del socket.

```
struct sockaddr
{
   unsigned short sa_family; /* familia de la dirección */
   char sa_data[14]; /* 14 bytes de la dirección del protocolo */
};
```

Pero, existe otra estructura, struct sockaddr_in, la cual nos ayuda a hacer referencia a los elementos del socket.

```
struct sockaddr_in
{
   short int sin_family; /* Familia de la Dirección */
   unsigned short int sin_port; /* Puerto */
   struct in_addr sin_addr; /* Dirección de Internet */
   unsigned char sin_zero[8];
   /* Del mismo tamaño que struct sockaddr */
};
```

Nota. sin_zero puede ser seteada con ceros usando las funciones memset() o bzero() (Ver los ejemplos).

La siguiente estructura no es muy usada pero está definida como una unión.

Como se puede ver en los dos ejemplos de abajo (ver la sección de nombre Un ejemplo de Servidor de Flujos y la sección de nombre Un ejemplo de Cliente de Flujos), cuando se declara, por ejemplo "client" para que sea del tipo sockaddr_in, luego se hace client.sin_addr = (...).

De todos modos, aquí está la estructura:

```
struct in_addr
{
   unsigned   long s_addr;
};
```

Finalmente, creemos que es mejor hablar sobre la estructura hostent. En el ejemplo de Cliente de Flujos (ver la sección de nombre Un ejemplo de Cliente de Flujos), se puede ver cómo se usa esta estructura, con la cual obtenemos información del nodo remoto.

Aquí se puede ver su definición:

```
struct hostent
{
   char *h_name; /* El nombre oficial del nodo.  */
   char **h_aliases; /* Lista de Alias.  */
   int h_addrtype; /* Tipo de dirección del nodo.  */
   int h_length; /* Largo de la dirección. */
```

```
   char **h_addr_list; /* Lista de direcciones del nombre del servidor. */
   #define h_addr  h_addr_list[0]  /* Dirección, para la compatibilidad con anteriores. */
};
```

Esta estructura está definida en el archivo netdb.h.

Al principio, es posible que estas estructuras nos confundan mucho. Sin embargo, luego de empezar a escribir algunas líneas de código, y luego de ver los ejemplos que se incluyen, será mucho más fácil entenderlas. Para ver cómo se pueden usar estas estructuras, recomiendo ver los ejemplos de la sección de nombre Un ejemplo de Servidor de Flujos y la sección de nombre Un ejemplo de Cliente de Flujos.

23.8.5-**Conversiones**

Existen dos tipos de ordenamiento de bytes: bytes más significativos, y bytes menos significativos. Este es llamado "Ordenamiento de Bytes para Redes'', y hasta algunas máquinas utilizan este tipo de ordenamiento para guardar sus datos, internamente.

Existen dos tipos a los cuales seremos capaces de convertir: short y long. Imaginémonos que se quiere convertir una variable larga de Ordenación de Bytes para Nodos a una de Ordenación de Bytes para Redes. ¿Qué haríamos? Existe una función llamada **htonl()** que hará exactamente esta conversión. Las siguientes funciones son análogas a esta y se encargan de hacer este tipo de conversiones:

htons() -> ``Nodo a variable corta de Red''

htonl() -> ``Nodo a variable larga de Red''

ntohs() -> ``Red a variable corta de Nodo''

ntohl() -> ``Red a variable larga de Nodo''

Una cosa importante, es que sin_addr y sin_port, de la estructura sockaddr_in, deben ser del tipo Ordenación de Bytes para Redes. Se verá, en los ejemplos, las funciones que aquí se describen para realizar estas conversiones, y a ese punto se entenderán mucho mejor.

23.8.6. **Direcciones IP**

En C, existen algunas funciones que nos ayudarán a manipular direcciones IP. En esta sección se hablará de las funciones inet_addr() y inet_ntoa().

Por un lado, la función inet_addr() convierte una dirección IP en un entero largo sin signo (unsigned long int), por ejemplo:

```
(...)

dest.sin_addr.s_addr = inet_addr("195.65.36.12");

(...)

/*Recordar que esto sería así, siempre que tengamos una
estructura "dest" del tipo sockaddr_in*/
```

Por otro lado, inet_ntoa() convierte a una cadena que contiene una dirección IP en un entero largo. Por ejemplo:

```
(...)

char *ip;

ip=inet_ntoa(dest.sin_addr);

printf("Address is: %s\n",ip);

(...)
```

Se deberá recordar también que la función inet_addr() devuelve la dirección en formato de Ordenación de Bytes para Redes por lo que no necesitaremos llamar a htonl().

23.8.7. **Funciones Importantes**

En esta sección, (en la cual se nombrarán algunas de las funciones más utilizadas para la programación en C de sockets), se mostrará la sintaxis de la función, las bibliotecas necesarias a incluir para realizar las llamadas, y algunos pequeños comentarios. Además de las que se mencionan aquí, existen muchas funciones más.

23.8.7.1. **socket()**

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain,int type,int protocol);
```

Analicemos los argumentos:

domain. Se podrá establecer como AF_INET (para usar los protocolos ARPA de Internet), o como AF_UNIX (si se desea crear sockets para la comunicación interna del sistema). Estas son las más usadas, pero no las únicas.

type. Aquí se debe especificar la clase de socket que queremos usar (de Flujos o de Datagramas). Las variables que deben aparecer son SOCK_STREAM o SOCK_DGRAM según queramos usar sockets de Flujo o de Datagramas, respectivamente.

protocol. Aquí, simplemente se puede establecer el protocolo a 0.

La función socket() nos devuelve un descriptor de socket, el cual podremos usar luego para llamadas al sistema. Si nos devuelve -1, se ha producido un error (obsérvese que esto puede resultar útil para rutinas de chequeo de errores).

23.8.7.2. **bind()**

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int fd, struct sockaddr *my_addr,int addrlen);
```

Analicemos los argumentos:

fd. Es el descriptor de archivo socket devuelto por la llamada a socket().

my_addr. es un puntero a una estructura sockaddr

addrlen. contiene la longitud de la estructura sockaddr a la cual apunta el puntero my_addr. Se debería establecerla como sizeof(struct sockaddr).

La llamada bind() es usada cuando los puertos locales de nuestra máquina están en nuestros planes (usualmente cuando utilizamos la llamada listen()). Su función esencial es asociar un socket con un puerto (de nuestra máquina). Análogamente socket(), devolverá -1 en caso de error.

Por otro lado podremos permitir que nuestra dirección IP y puerto sean elegidos automáticamente:

```
server.sin_port = 0;  /* bind() will choose a random port*/

server.sin_addr.s_addr = INADDR_ANY; /* puts server's IP automatically */
```

Un aspecto importante sobre los puertos y la llamada bind() es que todos los puertos menores que 1024 son reservados. Se podrá establecer un puerto, siempre que esté entre 1024 y 65535 (y siempre que no estén siendo usados por otros programas).

23.8.7.3. **connect()**

```
#include <sys/types.h>
```

```
#include <sys/socket.h>

int connect(int fd, struct sockaddr *serv_addr, int addrlen);
```

Analicemos los argumentos:

fd. Debería setearse como el archivo descriptor del socket, el cual fue devuelto por la llamada a socket().

serv_addr. Es un puntero a la estructura sockaddr la cual contiene la dirección IP destino y el puerto.

addrlen. Análogamente de lo que pasaba con bind(), este argumento debería establecerse como sizeof(struct sockaddr).

La función connect() es usada para conectarse a un puerto definido en una dirección IP. Devolverá -1 si ocurre algún error.

23.8.7.4. **listen()**

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int fd,int backlog);
```

Veamos los argumentos de listen():

fd. Es el archivo descriptor del socket, el cual fue devuelto por la llamada a socket()

backlog. Es el número de conexiones permitidas.

La función listen() se usa si se está esperando conexiones entrantes, lo cual significa, si se quiere, alguien que se quiera conectar a nuestra máquina.
Luego de llamar a listen(), se deberá llamar a accept(), para así aceptar las conexiones entrantes. La secuencia resumida de llamadas al sistema es:

1.socket()

2.bind()

3.listen()

4.accept()

Como todas las funciones descriptas arriba, listen() devolverá -1 en caso de error.

23.8.7.5. **accept()**

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int fd, void *addr, int *addrlen);
```

Veamos los argumentos de la función:

fd. Es el archivo descriptor del socket, el cual fue devuelto por la llamada a listen().

addr. Es un puntero a una estructura sockaddr_in en la cual se pueda determinar qué nodo nos está contactando y desde qué puerto.

addrlen. Es la longitud de la estructura a la que apunta el argumento addr, por lo que conviene establecerlo como sizeof(struct sockaddr_in), antes de que su dirección sea pasada a accept().

Cuando alguien intenta conectarse a nuestra computadora, se debe usar accept() para conseguir la conexión. Es muy fácil de entender: alguien sólo podrá conectarse (asóciese con connect()) a nuestra máquina, si nosotros aceptamos (asóciese con accept()) ;-)

A continuación, Se dará un pequeño ejemplo del uso de accept() para obtener la conexión, ya que esta llamada es un poco diferente de las demás.

```
(...)

sin_size=sizeof(struct sockaddr_in);
/* En la siguiente línea se llama a accept() */
if ((fd2 = accept(fd,(struct sockaddr *)&client,&sin_size))==-1){
printf("accept() error\n");
exit(-1);
}

(...)
```

A este punto se usará la variable fd2 para añadir las llamadas send() y recv().

23.8.7.6. **send()**

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int fd,const void *msg,int len,int flags);
```

Y sobre los argumentos de esta llamada:

fd. Es el archivo descriptor del socket, con el cual se desea enviar datos.

msg. Es un puntero apuntando al dato que se quiere enviar.

len. es la longitud del dato que se quiere enviar (en bytes).

flags. deberá ser es establecido a 0.

El propósito de esta función es enviar datos usando sockets de flujo o sockets conectados de datagramas.
Si se desea enviar datos usando sockets no conectados de datagramas debe usarse la llamada sendto(). Al igual que todas las demás llamadas que aquí se vieron, send() devuelve -1 en caso de error, o el número de bytes enviados en caso de éxito.

23.8.7.7. **recv()**

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(int fd, void *buf, int len, unsigned int flags);
```

Veamos los argumentos:

fd. Es el descriptor del socket por el cual se leerán datos.

buf. Es el buffer en el cual se guardará la información a recibir.

len. Es la longitud máxima que podrá tener el buffer.

flags. Por ahora, se deberá establecer como 0.

Al igual de lo que se dijo para send(), esta función es usada con datos en sockets de flujo o sockets conectados de datagramas. Si se deseara enviar, o en este caso, recibir datos usando sockets desconectados de Datagramas, se debe usar la llamada recvfrom(). Análogamente a send(), recv() devuelve el número de bytes leídos en el buffer, o -1 si se produjo un error.

23.8.7.8. **recvfrom()**

```
#include <sys/types.h>
```

```
#include <sys/socket.h>

int recvfrom(int fd,void *buf, int len, unsigned int flags
             struct sockaddr *from, int *fromlen);
```

Veamos los argumentos:

fd. Lo mismo que para recv()

buf. Lo mismo que para recv()

len. Lo mismo que para recv()

flags. Lo mismo que para recv()

from. Es un puntero a la estructura sockaddr.

fromlen. Es un puntero a un entero local que debería ser inicializado a sizeof(struct sockaddr).

Análogamente a lo que pasaba con recv(), recvfrom() devuelve el número de bytes recibidos, o -1 en caso de error.

23.8.7.9. **close()**

```
#include <unistd.h>

close(fd);
```

La función close() es usada para cerrar la conexión de nuestro descriptor de socket. Si llamamos a close() no se podrá escribir o leer usando ese socket, y se alguien trata de hacerlo recibirá un mensaje de error.

23.8.7.10. **shutdown()**

```
#include <sys/socket.h>

int shutdown(int fd, int how);
```

Veamos los argumentos:

fd. Es el archivo descriptor del socket al que queremos aplicar esta llamada.

how. Sólo se podrá establecer uno de estos nombres:
0. Prohibido recibir.
1. Prohibido enviar.
2. Prohibido recibir y enviar.

Es lo mismo llamar a close() que establecer how a 2. shutdown() devolverá 0 si todo ocurre bien, o -1 en caso de error.

23.8.7.11. **gethostname()**

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

Veamos de qué se tratan los argumentos:

hostname. Es un puntero a un vector que contiene el nombre del nodo actual.

size. La longitud del vector que contiene al nombre del nodo (en bytes).

La función gethostname() es usada para adquirir el nombre de la máquina local.

23.8.8. **Algunas palabras sobre DNS**

DNS son las siglas de ``Domain Name Service [9]'' y, básicamente es usado para conseguir direcciones IP. Por ejemplo, necesito saber la dirección IP del servidor queima.ptlink.net y usando el DNS puedo obtener la dirección IP 212.13.37.13.

Esto es importante en la medida de que las funciones que ya vimos (como bind() y connect()) son capaces de trabajar con direcciones IP.

Para mostrar cómo se puede obtener la dirección IP de un servidor, por ejemplo de queima.ptlink.net, utilizando C, el autor ha realizado un pequeño

ejemplo:

```
/* <---- EL CÓDIGO FUENTE EMPIEZA AQUÍ ----> */

#include <stdio.h>
#include <netdb.h>   /* Este es el archivo de cabecera necesitado por gethostbyname() */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{

   struct hostent *he;

   if (argc!=2) {
      printf("Usage: %s <hostname>\n",argv[0]);
      exit(-1);
   }

   if ((he=gethostbyname(argv[1]))==NULL) {
      printf("gethostbyname() error\n");
      exit(-1);
   }

   printf("Hostname : %s\n",he->h_name);
      /* muestra el nombre del nodo */
   printf("IP Address: %s\n",
         inet_ntoa(*((struct in_addr *)he->h_addr)));
      /* muestra la dirección IP */

}

/* <---- CÓDIGO FUENTE TERMINA AQUÍ ---> */
```

23.8.9. **Un ejemplo de Servidor de Flujos**

En esta sección, se describirá un bonito ejemplo de un servidor de flujos. El código fuente tiene muchos comentarios para que así, al leerlo, no nos queden dudas.

Empecemos:

```
/* <---- El CÓDIGO FUENTE COMIENZA AQUÍ ----> */

/* Estos son los ficheros de cabecera usuales */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 3550 /* El puerto que será abierto */
#define BACKLOG 2 /* El número de conexiones permitidas */

main()
```

```
{

   int fd, fd2; /* los archivos descriptores */

   struct sockaddr_in server;
   /* para la información de la dirección del servidor */

   struct sockaddr_in client;
   /* para la información de la dirección del cliente */

   int sin_size;

   /* A continuación la llamada a socket() */
   if ((fd=socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
      printf("error en socket()\n");
      exit(-1);
   }

   server.sin_family = AF_INET;

   server.sin_port = htons(PORT);
   /* ¿Recuerdas a htons() de la sección "Conversiones"? =) */

   server.sin_addr.s_addr = INADDR_ANY;
   /* INADDR_ANY coloca nuestra dirección IP automáticamente */

   bzero(&(server.sin_zero),8);
   /* escribimos ceros en el reto de la estructura */


   /* A continuación la llamada a bind() */
   if(bind(fd,(struct sockaddr*)&server,
           sizeof(struct sockaddr))==-1) {
      printf("error en bind() \n");
      exit(-1);
   }

   if(listen(fd,BACKLOG) == -1) {  /* llamada a listen() */
      printf("error en listen()\n");
      exit(-1);
   }

   while(1) {
      sin_size=sizeof(struct sockaddr_in);
      /* A continuación la llamada a accept() */
      if ((fd2 = accept(fd,(struct sockaddr *)&client,
                        &sin_size))==-1) {
         printf("error en accept()\n");
         exit(-1);
      }

      printf("You got a connection from %s\n",
             inet_ntoa(client.sin_addr) );
      /* que mostrará la IP del cliente */

      send(fd2,"Bienvenido a mi servidor.\n",22,0);
      /* que enviará el mensaje de bienvenida al cliente */

      close(fd2); /* cierraclose a fd2 */
   }
}

/* <---- EL CÓDIGO FUENTE TERMINA AQUÍ ----> */
```

23.8.10. **Un ejemplo de Cliente de Flujos**

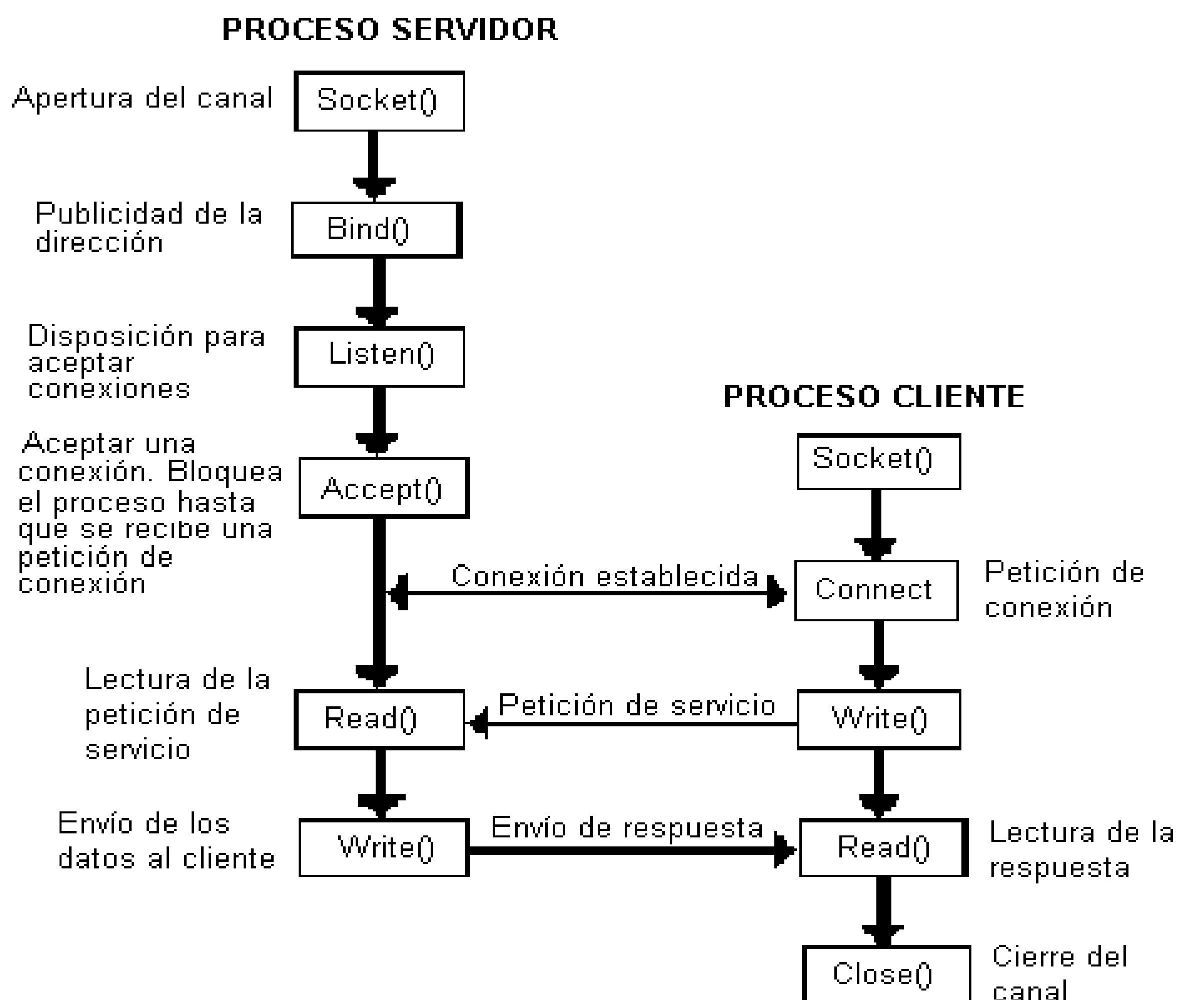
Todo será análogo a lo visto en la sección anterior.

```
/* <---- El CÓDIGO FUENTE COMIENZA AQUÍ ----> */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
/* netbd.h es necesitada por la estructura hostent ;-) */

#define PORT 3550
/* El Puerto Abierto del nodo remoto */

#define MAXDATASIZE 100
/* El número máximo de datos en bytes */

int main(int argc, char *argv[])
{
  int fd, numbytes;
  /* archivos descriptores */

  char buf[MAXDATASIZE];
  /* en donde es almacenará el texto recibido */

  struct hostent *he;
  /* estructura que recibirá información sobre el nodo remoto */

  struct sockaddr_in server;
  /* información sobre la dirección del servidor */

  if (argc !=2) {
    /* esto es porque nuestro programa sólo necesitará un
    argumento, (la IP) */
    printf("Usage: %s <IP Address>\n",argv[0]);
    exit(-1);
  }

  if ((he=gethostbyname(argv[1]))==NULL){
    /* llamada a gethostbyname() */
    printf("gethostbyname() error\n");
    exit(-1);
  }

  if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1){
    /* llamada a socket() */
    printf("socket() error\n");
    exit(-1);
  }

  server.sin_family = AF_INET;
  server.sin_port = htons(PORT);
  /* htons() es necesaria nuevamente ;-o */
  server.sin_addr = *((struct in_addr *)he->h_addr);
  /*he->h_addr pasa la información de ``*he'' a "h_addr" */
  bzero(&(server.sin_zero),8);

  if(connect(fd, (struct sockaddr *)&server,
    sizeof(struct sockaddr))==-1){
    /* llamada a connect() */
    printf("connect() error\n");
```

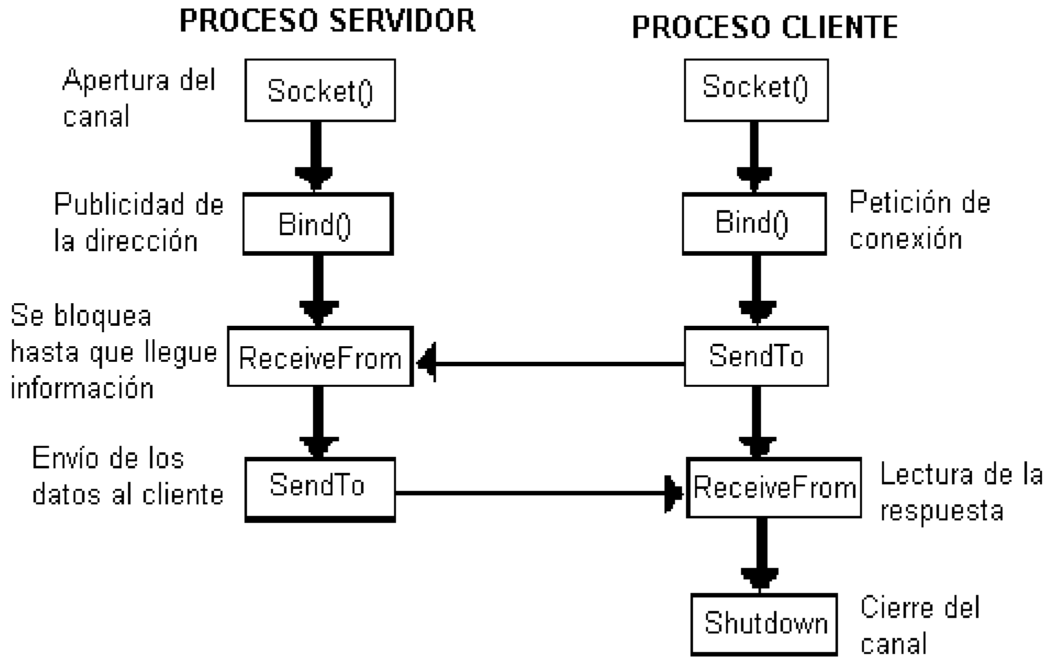
```
    exit(-1);
  }

  if ((numbytes=recv(fd,buf,MAXDATASIZE,0)) == -1){
    /* llamada a recv() */
    printf("recv() error\n");
    exit(-1);
  }

  buf[numbytes]='\0';

  printf("Server Message: %s\n",buf);
  /* muestra el mensaje de bienvenida del servidor =) */

  close(fd);   /* cerramos fd =) */

}

/* <---- EL CÓDIGO FUENTE TERMINA AQUÍ ----> */
```

En resumen podemos esquematizar la comunicación entre sockets, en modelo cliente/servidor de la siguiente manera:

Orientado a conexión:



Orientado a datagramas, o sea sin conexión:



23.9. - **INTERCAMBIO DE MENSAJES**.

La implementación de los mecanismos de comunicación y sincronización a través del intercambio de mensajes se da por el envío (**send**) y recepción (**receive**) de mensajes, en vez de la lectura o escritura de una variable compartida.

La comunicación ocurre porque una tarea al recibir un mensaje obtiene datos enviados por otra tarea, en tanto que la sincronización se da porque un mensaje sólo puede ser recibido después de haber sido enviado, lo que restringe el orden en el cual estos eventos deben ocurrir.

Ejemplo: **Send** mensaje **to** destino
Receive mensaje **from** origen

Asumimos como hipótesis que luego de recibido el mensaje, éste se destruye.

Estudiaremos el intercambio de mensajes por :

- Tipos de sincronización.
- Especificación de los canales de comunicación.
- Tipos de mensajes.
- Tratamiento y recuperación de errores.

Para este fin utilizaremos las siguientes primitivas:

Send sincrónico : la tarea emisora queda bloqueada hasta que el mensaje sea recibido por la tarea destino.
Send asincrónico : la tarea emisora luego de emitir el mensaje continua con su procesamiento. De esta forma la concurrencia entre las tareas que se comunican es maximizada.
Send condicional: la tarea emisora luego de emitir el mensaje continúa con su procesamiento, pero el mensaje solamente será recibido por la tarea receptora si es que ésta se encuentra bloqueada en el momento de emisión del mensaje. En caso contrario la tarea emisora recibe un código de retorno que le indica que el mensaje no ha llegado a destino. La utilización de esta primitiva implica que la tarea receptora utilice un receive bloqueante.
Receive (incondicional o bloqueante): la tarea receptora queda bloqueada hasta recibir un mensaje.
Receive condicional (polling): la tarea receptora pregunta a los canales de comunicación si existe un mensaje. En caso negativo, la primitiva receive devuelve un código de retorno que indica que no hay mensajes en ese canal de comunicación. Esta capacidad de consulta (polling) permite que la tarea receptora controle el nivel de concurrencia.

23.9.1. - **TIPOS DE SINCRONIZACION.**

Las primitivas de comunicación pueden clasificarse en sincrónicas y asincrónicas según bloqueen o no a las tareas que ejecutan la emisión o recepción de un mensaje.

23.9.1.1. - **Comunicación Sincrónica.**

Los mecanismos de intercambio de mensajes basados en el uso de primitivas de comunicación y sincronización sincrónica se clasifican en tres categorías discutidas a continuación:

a) Rendez-Vous

La tarea emisora es bloqueada hasta que la tarea receptora esté lista para recibir el mensaje. Cuando la tarea receptora ejecuta un receive, si no se encuentra disponible un mensaje entonces queda bloqueada hasta la llegada del mismo.

Una vez efectuado el intercambio de mensajes ambas tareas continúan su ejecución en forma concurrente.

```
tarea Productor;
    begin
    <Producir un msg>;
    Send msg to Consumidor
    end;
tarea Consumidor;
    begin
    Receive msg from Productor;
    <Consumir msg>
    end;
```

Una abstracción utilizando una implementación con semáforos sería:

```
V(x)
P(y)
SEND

P(x)
V(y)
RECEIVE
```

Obsérvese que en este caso las velocidades de producción y consumo de mensajes son equivalentes debido a la sincronización explícita de las tareas.

b) Rendez-Vous extendido

Como su nombre lo indica, esta forma de interacción entre tareas es una extensión del mecanismo anterior, con la diferencia de que la tarea receptora solamente envía una respuesta a la tarea emisora después de la ejecución de un cuerpo de comandos que operan sobre el mensaje recibido. Esta respuesta puede poseer parámetros que contengan el resultado de los cálculos efectuados por la tarea receptora. Obsérvese también que la tarea emisora queda bloqueada hasta el término del cuerpo de comandos en la tarea receptora, que es cuando el rendez-vous se completa.

Con semáforos se resume en:

```
V(x)
SEND
P(y)

P(x)
RECEIVE
If ok Then V(y)
```

c) Rendez-Vous asimétrico

Es una variante del anterior. Aquí solamente el emisor (cliente) nombra a la tarea receptora (server) (ADA).

La primitiva receive es reemplazada por el comando **accept**. Ambas tareas quedan en rendez-vous hasta que se ejecute todo el cuerpo del comando accept. El accept no nombra al emisor pues la comunicación ya está establecida.

Los parámetros (ya que es como llamar a un monitor) pueden ser de input, de output o de input / output.

```
Tarea T1;
    begin
    Send x to T2;
    end;
```

```
Tarea T2;
        begin
        Accept Send (x);
        y := x;
        end;
```

23.9.1.2. - **Comunicación Asincrónica.**

Las primitivas de comunicación asincrónica se caracterizan por no bloquear a las tareas que las ejecutan.
Una tarea emisora al realizar un send asincrónico continua su ejecución sin bloquearse.
En el caso de receive asincrónico el receptor continúa su ejecución aunque no haya llegado nada. Depende de la implementación si los mensajes siguientes serán o no tomados en cuenta (Ej. Spool de VM).
Este tipo de envío no bloqueante permite la múltiple difusión de mensajes (**broadcasting**), o sea, que un mismo mensaje es enviado a varios destinatarios simultáneamente.
La principal ventaja de la comunicación asincrónica es que maximiza el paralelismo en la ejecución de las tareas.

23.9.1.3. - **Comunicación Semi-Sincrónica.**

Una variante a los esquemas de comunicación sincrónica y asincrónica es la comunicación semi-sincrónica que usa **send** no bloqueantes y **receive** bloqueantes. Sin embargo en esta implementación se corre el peligro de tener largas colas de mensajes.

23.9.2. - **Especificaciones de los Canales de Comunicación**

Definir Origen y Destino de los mensajes define un canal de comunicación.
Existen dos tipos de especificaciones, a saber:

a) Comunicación directa

Cada proceso que desea enviar/recibir un mensaje debe nombrar explícitamente el receptor/emisor del mensaje en cuestión.
Un proceso para comunicarse con otro sólo debe conocer la identidad del otro proceso.
Cada enlace de comunicación de este tipo vincula a dos procesos y es bidireccional.
El lenguaje CSP utiliza los comandos:

- de entrada P1 ? A
- de salida P2 ! B

siendo P1 y P2 las tareas.
Este esquema se usa cuando la salida de una tarea es entrada de otra (Pipeline (Unix)).
Sin embargo, este mecanismo no resuelve el problema del server que atiende a más de un cliente, o el cliente que llame a más de un server.

b) Comunicación indirecta

Este mecanismo soluciona el problema anterior mediante el uso de nombres globales (Mailboxes).
Los procesos envían y reciben mensajes desde mailboxes. Un mailbox puede ser visto como un objeto en el que los mensajes son depositados y retirados por los procesos. Cada mailbox tiene una identificación que lo determina unívocamente.
Dos procesos pueden comunicarse sólo si comparten un mailbox.
La comunicación entre procesos puede vincular a más de dos procesos y dos procesos se pueden comunicar a través de varios mailboxes. La comunicación entre procesos puede ser unidireccional o bidireccional.
Esta mecánica funciona cuando, por ejemplo, se comparte memoria, pero no funciona para sistemas distribuidos a menos que se le atribuya al mailbox un mecanismo de red. Esto implica que se deben colocar estos nombres globales en todos los puntos de la red donde ese mensaje podría ser usado.
Un caso especial es aquel que se establece cuando una única tarea está autorizada a hacer receives en un mailbox pero varias tareas pueden hacer sends a dicho mailbox. Este tipo de mailbox se denomina **puerta** (port). Este esquema es más fácil de implementar porque todos los receives que pueden referenciar a una misma puerta se hallan dentro de una única tarea.

23.9.3. - **Direccionamiento**

En resumen los direccionamientos pueden ser:

Directo: Dos tareas sobre nodos distintos mantienen comunicación directa entre sí.

Indirecto: La comunicación se mantiene a través de una estructura de datos compartida (llamada mailbox)
Ventaja: desacopla al emisor y al receptor, permitiendo mayor flexibilidad en el uso de los mensajes.
Desventaja: Centraliza.

La utilización del mailbox es posible en las modalidades:

- Uno a uno, también llamado enlace privado.
- Muchos a uno, que corresponde al modelo cliente/servidor.
- Uno a muchos, también llamado broadcast.
- Muchos a muchos, que se puede visualizar como la posibilidad de tener muchos servidores

Por lo general los mailbox manejan disciplinas de colas en modalidad FIFO, pero también es posible el manejo por prioridades.

23.9.4. - **Ejemplo de Modelos Clásicos con mailbox**

23.9.4.1. - **Exclusión Mutua**

Se utilizan **receive bloqueantes y send no bloqueantes**.
El **mailbox** se lo utiliza como contenedor de un **token.**

```
/* programa exclusion-mutua */
int  n=   /* número de procesos */

void p(int i)
{
        mensaje msj;
        while (cierto)
        {
                receive (exmut, msj);  /*si el mailbox está vacío el proceso se detiene */
                /* sección crítica */
                send (exmut, msj);
                /* resto */
        }
}

void main ()
{
        crear-mailbox (exmut);
        send (exmut, token);
        parbegin (p1, p2, p3, ..., pn);
}
```

23.9.4.2. - **Productor/Consumidor**

Se utilizan **receive bloqueantes y send no bloqueantes**.
Se utilizan dos buzones, puede_consumir y puede_producir

```
Capacidad =     /* capacidad del buffer */;
Int i;

Void productor()
{
        mensaje msjp;
        while (cierto);
        {
                receive (puede_producir, msjp);
                msjp = producir();
                send (puede-consumir, msjp);
        }
}
```

```
void consumidor()
{
	mensaje msjc;
	while (cierto)
	{
		receive (puede_consumir, msjc);
		consumir(msjc);
		send (puede_producir, token);
	}
}

void main()
{
	crear_mailbox (puede_producir);
	crear_mailbox (puede_consumir);
	for (int i = 1, i <= capacidad; i++)
		send (puede_producir, token);
		send (puede_consumir, null);
	parbegin (productor, consumidor) parend;
}
```

23.9.4.3. - **Lectores/Escritores**

Se utilizan **3 mailbox pedir_lectura pedir-escritura** y **terminado.**
Además de los procesos lector y escritor se utiliza uno auxiliar llamado controlador que actúa según el valor de una variable **cont, según lo siguiente:**

Cont > 0 no hay escritores esperando
Cont = 0 pendientes escrituras, esperar terminado
Cont < 0 escritor en espera
Cantidad máxima de lectores = 100

```
void lector(int i)
{
	mensaje msjl;
	while (cierto)
	{
		msjl = i;
		send (pedir-lectura, msjl);
		receive (buzón[i], msjl);
		LEER;
		msjl = i;
		send (terminado, msjl);
	}
}


void escritor(int j)
{
	mensaje msje;
	while (cierto)
	{
		msje=j;
		send (pedir_escritura, msje);
		receive (buzón[j], msje);
		ESCRIBIR;
		msje = j;
		send (terminado, msje)
	}
}
```

```
void controlador()
{
	while(cierto)
	{
		if cont > 0
		{
			if (!vacío (terminado))
			{
				receive (terminado, msj);
				cont++
			}
			else
			if (!vacío (pedir_escritura))
			{
				receive (pedir_escritura, msj);
				escritor_id = msj-id;
				cont  = cont – 100;
			}
			else
			if (!vacío (pedir_lectura))
			{
				receive (pedir_lectura, msj));
				cont--;
				send(msj_id, “OK”);
			}
		}
		if (cont == 0)
		{
			send (escritor_id, “OK”);
			receive (terminado, msj);
			cont = 100;
		}
		while (cont < 0)
		{
			receive (terminado, msj);
			cont ++;
		}
	}
}
```

23.9.5. - **Tipos de Mensajes**.

Los mensajes enviados por un proceso pueden ser de tres tipos: tamaño fijo, tamaño variable o mensajes tipificados.

Si solamente se pueden enviar mensajes de tamaño fijo, la implementación física es bastante sencilla. Esta restricción, sin embargo, hace a la tarea de programación más difícil.

Por otro lado, los mensajes de longitud variable requieren una implementación física más compleja pero facilitan la programación.

El último caso consiste en asociar a cada mailbox un tipo de mensaje, por lo que este esquema es sólo aplicable cuando se usa comunicación indirecta. Esta asociación permite detectar, en tiempo de compilación, la posibilidad del intercambio de mensajes de diferente tipo brindando un alto grado de seguridad sin provocar sobrecarga en tiempo de ejecución.

23.9.6. - **Tratamiento y Recuperación de Errores.**

Existen dos tipos de fallas que merecen un tratamiento adecuado para garantizar, que cuando ocurra alguna de ellas, sea posible preservar la integridad del sistema.

El primer tipo de falla se denomina falla lógica y se da cuando se produce un deadlock o cuando una tarea envía o espera recibir un mensaje de otra tarea que ya no existe (destrucción prematura de tareas).

El otro tipo de falla se denomina falla física y se da cuando trabajamos en un sistema distribuido y la falla se produce en el hardware de comunicación.

Para ambos tipos de errores es necesario contar con un mecanismo de time-out para evitar el problema del bloqueo perpetuo.

CAPITULO 24

THREADS

24.1. - **RPC CON HILOS**

La idea de usar hilos en RPC es aligerar a este último.

RPC local

Al iniciar un hilo servidor S, éste exporta su interfase informándosela al núcleo. La interfase define los procedimientos que puede llamar, sus parámetros, etc. Al iniciar un hilo cliente C este importa la interfase del núcleo y se le proporciona un identificador especial para utilizarlo en la llamada, entonces el núcleo sabe que C va a llamar a S, y crea estructuras de datos especiales con el fin de prepararse para la llamada. Una de estas estructuras es la pila compartida entre C y S y que se asocia de manera lectura/escritura en ambos espacios de direcciones.

Para llamar al servidor, C coloca sus argumentos en dicha pila utilizando para ello el procedimiento normal de transferencia y después hace una interrupción al núcleo al colocar un identificador especial en un registro. El núcleo se da cuenta de esto y sabe que es una llamada local, desarrollando las siguientes acciones: Modifica el mapa de memoria del cliente para colocarlo en el espacio de direcciones del servidor y dentro de este inicializa el hilo cliente al ejecutar el procedimiento del servidor. Esto se realiza de forma tal que los argumentos se encuentran ya en su lugar, o sea que no necesitan ser copiados.

Otra forma de agilizar el RPC local es haciendo desvanecer un hilo servidor (desaparece su pila e información de contexto) cuando termina de realizar una solicitud ya que es raro que deba tener variables locales y los datos en sus registros no tienen significado ya.

Recepción Implícita

Al llegar un nuevo mensaje a la máquina servidora, el núcleo crea en ese momento un nuevo hilo para darle servicio a la solicitud, además asocia el mensaje con el espacio de direcciones del servidor y configura la pila del nuevo hilo para tener acceso al mensaje (hilo de aparición instantánea - pop-up thread).

En este método los hilos no tienen que bloquearse en espera de más trabajo y por lo tanto no es necesario guardar información del contexto. Además la creación de un nuevo hilo es más económica que la restauración de uno ya existente y se ahorra tiempo al no tener que copiar los mensajes recibidos a un buffer dentro de un hilo servidor.

24.2. - **MODELOS DE SISTEMAS**

Los procesadores de un sistema distribuido se pueden organizar de distintas formas. A continuación veremos tres formas que se basan en diferentes filosofías respecto de lo que debe ser un sistema distribuido.

24.2.1. - El Modelo de Estación de Trabajo

Este modelo es directo, o sea el sistema consta de estaciones de trabajo, computadoras personales de alta calidad dispersas en un edificio y conectadas entre sí por medio de una LAN de alta velocidad. Las estaciones de trabajo pueden tener más de un usuario, pero solo uno a la vez puede estar conectado a ella o la estación puede estar inactiva.

Existen dos tipos de estaciones de trabajo, con disco y sin disco. Si la estación de trabajo no tiene disco el sistema de archivos debe manejarse por medio de uno o más servidores de archivos en la red.

Los usuarios tienen una cantidad fija de poder de cómputo exclusivo, garantizando un tiempo de respuesta. Cada usuario tiene un alto grado de autonomía y puede asignar los recursos de su estación de trabajo como juzgue sea necesario.

Una desventaja es que existen baches en que los usuarios no utilizan las máquinas y es probable que otros usuarios necesiten una capacidad de cómputo mayor y no puedan obtenerla. Una solución es usar estas estaciones de trabajo inactivas.

a) Si las estaciones de trabajo carecen de disco: el sistema de archivos debe ser implementado mediante uno o más servidores de red y las solicitudes de lectura/ escritura serán enviadas al servidor de archivos que realiza el trabajo y envía luego las respuestas. Tienen fácil mantenimiento y menor precio. Además, proporcionan simetría y flexibilidad.

b) Si las estaciones tienen sus propios discos: estos pueden ser usados de cuatro maneras:

- i) Paginación y archivos temporales: Aunque es conveniente mantener los archivos de un usuario en servidores centrales de archivo para facilitar el respaldo y mantenimiento, también se necesitan discos para paginación y archivos temporales, si se usa el disco local se reduce el tráfico en la red.
- ii) Paginación, archivos temporales y códigos objeto del sistema: Los discos locales también contienen programas ejecutables como ser compiladores, editores y manejadores de correo electrónico.

Al disponer una nueva versión del programa se transmite a todas las máquinas, pero si una máquina estaba apagada en ese instante queda con la versión vieja.

iii) Paginación, archivos temporales, códigos objeto del sistema y ocultamiento de archivos: Utiliza los discos locales como cache durante el acceso a un archivo del servidor, además de usarlo para paginación, archivos temporales y códigos binario. Así los usuarios pueden cargar archivos desde los servidores hasta sus propios discos, trabajarlos localmente y luego restaurarlos. Con esto se logra mantener centralizado el almacenamiento a largo plazo. Pero tiene un problema cuando dos usuarios utilizan el mismo archivo.

iv) Sistema local de archivo completo: Cada máquina puede tener su propio sistema de archivos autocontenido, pudiendo montar y tener acceso a los sistemas de archivos de otras máquinas. El objetivo de esto es que cada máquina esté autocontenida con un limitado contacto con el mundo exterior. Esto conlleva una pérdida de transparencia. El sistema resultante se parece más a un sistema operativo de red que a un sistema distribuido.

Uso del Disco	Ventajas	Desventajas
Sin disco	Bajo costo, fácil mantenimiento del hardware y el software, simetría y flexibilidad	Gran uso de la red, los servidores de archivos se pueden convertir en cuellos de botella
Paginación, archivos de tipo borrador	Reduce la carga de la red comparada con el caso sin disco	Un costo alto debido al gran número de discos necesarios
Paginación	Reduce todavía más la carga sobre la red	Alto costo, complejidad adicional para actualizar los códigos objeto
Paginación, archivos de tipo borrador, códigos objeto, ocultamiento de archivos	Una carga aún menor en la red, también reduce la carga en los servidores de archivos	Alto costo, problemas de consistencia del cache
Sistema local de archivos completo	Escasa carga en la red, elimina la necesidad de los servidores de archivos	Pérdida de transparencia

24.2.1.1. - Uso de estaciones de trabajo inactivas

Uno de los intentos de tratar de aprovechar la potencia de cómputo de una estación de trabajo inactiva consiste en el uso del Remote Login (RSH).

rsh máquina comando

Esto tiene varios problemas:

1) **Como localizar una estación de trabajo inactiva ?**

Una estación donde ningún usuario está conectado puede ejecutar decenas de procesos (demonios de correo, noticias, reloj, etc.), más aún, un usuario puede estar conectado pero si hace varios minutos que no toca el teclado o si la máquina no está ejecutando algún proceso en concreto se puede decir que dicha máquina se encuentra inactiva.

Los algoritmos para localizar estas estaciones se dividen en dos:

Controladas por servidor : cuando una estación de trabajo esta inactiva anuncia su disponibilidad nombre, dirección y propiedades en un archivo de registro en el servidor o coloca un mensaje que se anuncia en toda la red, registrando esto todas las otras estaciones. No obstante ello en este caso puede suceder que dos estaciones que buscan una inactiva detecten la misma y traten de iniciar un proceso al mismo tiempo.

Controladas por cliente : el cliente transmite una solicitud donde indica el programa que desea ejecutar y la cantidad de memoria necesaria, si necesita coprocesador, etc. (esto no es necesario si las estaciones son iguales). Todas las que están libres envían un mensaje a la que solicitó y ella decide con cuál trabajar. Si el mensaje de respuesta se demora directamente proporcional a la carga de la máquina que responde las respuestas de las máquinas con menor carga llegarán antes.

2) **Como lograr que un proceso remoto se ejecute en forma transparente ?**

Necesita la misma visión del sistema de archivos, el mismo directorio de trabajo, las mismas variables del ambiente si existen para poder ejecutar en la remota. Algunas de las llamadas al sistema se deben devolver a la máquina de origen sin hacer distinción alguna, otras nunca se pueden ejecutar en la máquina remota como ser lectura del teclado. Las llamadas al sistema relacionadas con el tiempo son un inconveniente puesto que los relojes de las distintas máquinas no tienen porqué estar sincronizados.

3) **Que ocurre si el usuario de la máquina inactiva regresa ?**

Lo más fácil es no hacer nada, pero esto va en contra de la idea de estaciones personales de trabajo. Otra posibilidad es eliminar el proceso intruso de manera abrupta y sin previo aviso (pérdida de trabajo y sistema de archivos caótico). Es mejor darle al proceso una advertencia y hacer esto con bondad. Otra solución es hacer que el proceso migre a otra máquina, ya sea a la máquina de origen o alguna otra inactiva. Esto último es posible pero tiene dificultades prácticas sustanciales, como por ejemplo lograr que migre el proceso padre y además todos sus hijos, debe llevarse sus buzones, conexiones de red, etc.

24.2.2. - El Modelo de la Pila de Procesadores

El problema que trata de resolver este esquema es cómo proveer a los usuarios de la potencia de cómputo de más de un CPU.

Lo que se hace es construir una pila de procesadores en el cuarto de máquinas los cuales se pueden asignar a los usuarios en forma dinámica. Los usuarios usan terminales gráficas de alto rendimiento, como las terminales X.

Desde el punto de vista conceptual se parecen mucho más al tiempo compartido tradicional que al modelo de computadora personal.

Si el sistema de archivos se debe concentrar en un pequeño número de servidores de archivos, debe ser posible hacer lo mismo con los servidores de cómputo. De hecho se convierte todo el poder de cómputos en " Estaciones de trabajo inactivas " a las que se puede tener acceso en forma dinámica. Este modelo proviene de las teorías de colas.

El estudio nos dice que si reemplazamos N pequeños recursos por uno grande que sea N veces mayor podemos reducir el tiempo promedio de respuesta N veces. Este resultado de la teoría de colas es uno de los principales argumentos en contra de los sistemas distribuidos. Los S.D. tienen a favor el costo que será mucho menor, también la confiabilidad y la tolerancia a fallas. Además de esto las estaciones de trabajo tienen una respuesta uniforme independientemente de lo que hagan las demás personas.

Además esta prestación del modelo de pila de procesadores se basa en que una solicitud se puede dividir para ejecutarse en los N procesadores de la pila si la solicitud solo puede partirse entre 3 procesadores el desempeño solo mejorará en un tercio.

No obstante, este modelo es mejor que el trabajo de buscar estaciones inactivas.

Por último la elección depende en gran medida del tipo de tareas que se realicen. Si los usuarios trabajan con edición de archivos y envían cada tanto algún mensaje de correo, probablemente baste con el modelo de estaciones de trabajo, en tanto que si un grupo de usuarios trabaja en el desarrollo de un gran proyecto de software, o hacen grandes simulaciones, probablemente será mejor el esquema de pila de procesadores.

24.2.3. - Un Modelo Híbrido

Se puede tener una estación de trabajo personal para cada usuario y además tener una pila de procesadores dedicada a los procesos no interactivos y a todo el cómputo pesado. No se realiza el trabajo de búsqueda de estaciones inactivas lo que hace más sencillo el diseño del sistema.

Si bien este esquema es el más caro combina las ventajas de los dos anteriores.

24.3. - **COMO SE ASIGNAN LOS PROCESADORES?**

El problema a tratar aquí es cómo asignar un procesador a un proceso. Por ejemplo en el modelo de estaciones de trabajo cuando asignar el procesador local o cuándo buscar una máquina inactiva; en el modelo de pila cómo asignar los procesadores cuando aparece un nuevo proceso.

Las estrategias de asignación de procesadores se pueden dividir en dos categorías :

No migratoria: Al crearse un proceso se decide donde ponerlo y permanece allí hasta que termina.

Migratorios: Un proceso puede trasladarse aunque haya iniciado su ejecución.

Evidentemente la idea subyacente a la asignación de procesadores es la optimización de algún componente del sistema. Sin embargo lo que se desee optimizar variará de un sistema a otro.

Una optimización posible es el **uso de la CPU**, es decir evitar a toda costa el tiempo ocioso de la misma.

Otro parámetro a tener en cuenta es **el tiempo promedio de respuesta** o su variante, **la tasa de respuesta** que se define como la cantidad de tiempo necesaria para ejecutar un proceso en cierta máquina dividido por el tiempo que tardaría en ejecutarse en un procesador de referencia no cargado.

Aspectos de diseño

Para diseñar estos algoritmos los diseñadores deben tomar decisiones con respecto a:

1) Algoritmos Deterministas vs. Heurísticos: Cuando se tiene de antemano toda la información sobre el comportamiento de los procesos se utilizan algoritmos determinísticos en tanto que los heurísticos se usan cuando la carga es impredecible.

2) Algoritmos Centralizados vs. Distribuidos: Se han propuesto algoritmo centralizados por la carencia muchas veces de buenas alternativas distribuidas.

3) Algoritmos óptimos vs. Subóptimos: Obtener el óptimo implica mayor complejidad de cálculo. La mayoría de los sistemas distribuidos reales buscan soluciones subóptimas, heurísticas y distribuidas debido a la dificultad para obtener las óptimas.
4) Algoritmos Locales vs. Globales: Para estos algoritmos se debe tener en cuenta que al crearse un proceso si hay que tomar la decisión de ejecutarlo en otra máquina (**política de transferencia**) la opción consiste en basar o no esta decisión por completo en la información local. Los algoritmos locales son sencillos pero están muy lejos de ser los óptimos, mientras que los globales sólo dan un resultado un poco mejor a un mayor costo.
5) Algoritmos Iniciados por el Receptor vs Iniciados por el Emisor: Una vez que se decidió liberarse de un proceso la **política de localización** debe decidir dónde enviarlo. Esta política no puede ser local, necesita información de la carga en todo el sistema. Esta información se puede dispersar como inicio de intercambio por parte de los emisores o por parte de los receptores.

24.4. - **PLANIFICACIÓN EN SISTEMAS DISTRIBUIDOS**

Por lo general cada procesador hace su propia planificación local, sin embargo, si un grupo de procesos relacionados entre sí y con una gran interacción, se ejecutan en distintos procesadores la planificación independiente no es eficiente.

Lo que se necesita es una forma de garantizar que los procesos con comunicación frecuente se ejecuten en forma simultánea.

Aunque es difícil determinar en forma dinámica los patrones de comunicación entre los procesos, en muchos casos, un grupo de procesos relacionados entre sí se iniciarán juntos. Supongamos que los procesos se crean en grupos y que la comunicación dentro de los grupos prevalece sobre la comunicación entre los grupos, y que se tiene un número de procesadores lo bastante grande como para manejar al grupo de mayor tamaño y que cada procesador se multiprograma con N espacios para los procesos.

Existe un concepto llamado COPLANIFICACION (Ousterhout 1982) el cual toma en cuenta los patrones de comunicación entre los procesos durante la planificación para garantizar que todos los miembros de un grupo se ejecuten al mismo tiempo.

Existen algoritmos para planificar esto:

Uno se realiza mediante una matriz con N entradas para los procesadores (columnas) y M para los espacios de tiempo. Así la columna 4 tiene todos los procesos que se ejecutan en el procesador 4 a través de los tiempos, el renglón 3 es la colección de todos los procesos en el espacio 3 para algún procesador.

Los procesadores ejecutan el proceso del espacio 0 durante un cierto tiempo luego todos los del 1 y así siguiendo usando un algoritmo round-robin. Se puede utilizar un mensaje para indicarle a cada procesador el momento en que debe intercambiar los procesos y mantener sincronizados los intervalos de tiempo.

Si todos los miembros de un grupo se colocan en el mismo número de espacio pero en procesadores distintos se obtiene paralelismo de nivel N, ejecutándose todos al mismo tiempo y maximizando el desempeño de la comunicación.

Una variante para mejorar sería separar la matriz por renglones y concatenarlos formando un gran renglón. Con k procesadores, k entradas cualquiera consecutivas pertenecen a distintos procesadores. Para asignar un nuevo grupo de procesos a las entradas se deja una ventana de k entradas de ancho en el renglón de gran tamaño de modo que la entrada del extremo izquierdo esté vacía, pero que la entrada justo a la izquierda de la ventana esté ocupada. Si existe el número suficiente de entradas en dicha ventana, los procesos se asignan a las entradas vacías o bien la ventana se desliza hacia la derecha y se repite el algoritmo.

La planificación se lleva a cabo, al iniciar la ventana en la orilla izquierda y moviéndola a la derecha, tantas entradas como tenga la ventana por cada intervalo de tiempo, teniendo cuidado de no dividir los grupos en las ventanas.

CAPITULO 25

DISTRIBUTED FILE SYSTEM

25.1. - Introducción.

En cualquier sistema informático y sus aplicaciones es necesario poder mantener y recuperar la información. Para esto es necesario organizarla y administrarla, con ese fin se provee como solución el almacenamiento en unidades denominadas archivos (files) en discos y otras unidades externas.

En el diseño de cualquier sistema operativo la administración de ellos es una de las partes más trascendentales. En el caso de un sistema operativo centralizado no es tan complicada dicha administración. Sin embargo, en uno distribuido no ocurre lo mismo.

En el caso de un sistema distribuido es importante diferenciar dos conceptos fundamentales:

Servicio de archivos: Es la especificación de los servicios que el file system provee a sus clientes, comprendiendo:

- Primitivas disponibles.
- Parámetros que utilizan dichas primitivas.
- Acciones que las primitivas proveen.

Para los clientes especifica claramente con qué servicios pueden contar, pero no dice nada acerca de cómo se lo implementa. Es decir, el servicio de archivos especifica la interfase del sistema de archivos de los clientes.

Servidor de archivos (de aquí en adelante server): Es un proceso que se ejecuta en alguna máquina contribuyendo a la implementación propia del servicio de archivos.

Si bien un sistema puede tener varios servers, los clientes no deberán conocer ni la cantidad de ellos ni su ubicación, es decir se deberá mantener la transparencia en el diseño e implementación de un file system.

Un sistema puede contener varios servidores de archivos (por ejemplo, un proceso de usuario ejecutándose en el kernel de una máquina) y cada uno puede ofrecer un servicio de archivos diferente.

25.2. - DISEÑO DE LOS FILE SYSTEMS

En general existen dos componentes:

- El servicio propio de archivos: Encargándose de las operaciones en los archivos individuales (creación, borrado, etc.).
- El servicio de directorios: Realiza la administración a nivel de directorios (crear, eliminar directorios, etc.).

25.3. - EL SERVICIO DE ARCHIVOS

Qué es un archivo? En un sistema como UNIX o DOS un archivo es una secuencia de bytes sin interpretación alguna. El significado y estructura de la información queda a cargo del programa de aplicación que la accede.

En sistemas mainframes existen muchos tipos de archivos con diferentes propiedades. En general se especifica el registro del archivo por medio de un número que es su posición dentro del archivo o por el valor de algún campo. En el segundo caso (DOS) el archivo se construye como un árbol u otra estructura como ser una tabla que indica la ubicación (usualmente dispersa) de los distintos registros.

La mayoría de los sistemas distribuidos soporta el concepto de archivo como secuencia de bytes en lugar de una secuencia de registros con cierta clave.

Otro aspecto importante es si los archivos se pueden modificar después de creados. Lo normal es que si, pero en algunos sistemas distribuidos las únicas operaciones sobre archivos son CREATE y READ. Estos archivos se denominan **inmutables**. Este tipo de archivo facilita el soporte para ocultamiento (caching) y replicación de archivos, ya que elimina la necesidad de actualizar todas las copias del archivo.

La protección se implementa con los mismos métodos estudiados anteriormente :

- listas de capacidades, o
- listas de control de acceso

Por ejemplo UNIX con los bits que controlan la lectura, escritura y ejecución para el dueño, el grupo del dueño y otros usuarios es una lista de accesos simplificada.

Los servicios de archivos se pueden dividir en dos tipos según la forma de acceso a los archivos, el modelo Upload/Download y el modelo de Montaje remoto.

25.3.1. - Upload/Download Model

Las operaciones básicas provistas (lectura y escritura de un archivo) se basan en el concepto del traslado por completo de la información, pudiendo almacenarse el archivo en memoria del cliente o en su un disco en forma local.

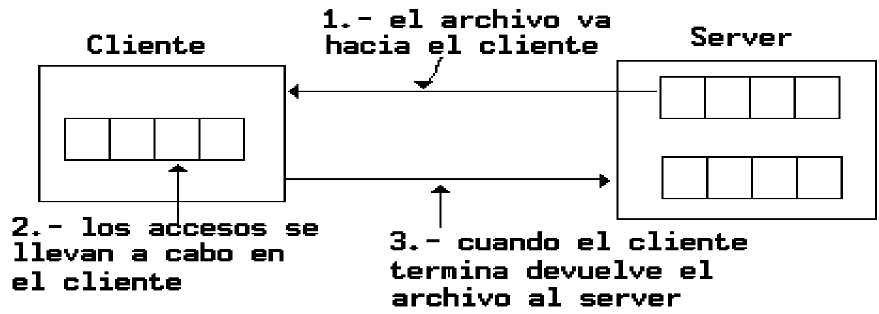


Fig. 25.1. - Modelo Upload/Download.

25.3.2. - Remote Access Model (montaje remoto).

Aquí lo que se realiza son solicitudes al server que es donde el archivo permanece sin realizarse la transferencia por completo de la información, sino que solamente se opera con el archivo a través de operaciones que se proveen.

Las solicitudes que se realizan tienen una gran variedad de posibilidades, por ejemplo: abrir y cerrar un archivo, leer y escribir partes de un archivo, moverse a través de un archivo, examinar y modificar atributos del archivo, etc.

Las ventajas de ambos métodos son: la sencillez del Upload/Download ya que la aplicación sólo busca los archivos que necesita y los utiliza de manera local, en tanto que en el Remote Access Model existe muy poca necesidad de espacio en los clientes ya que no transfiere los archivos en forma completa sino que solo lo hace con aquellas partes que precisa.

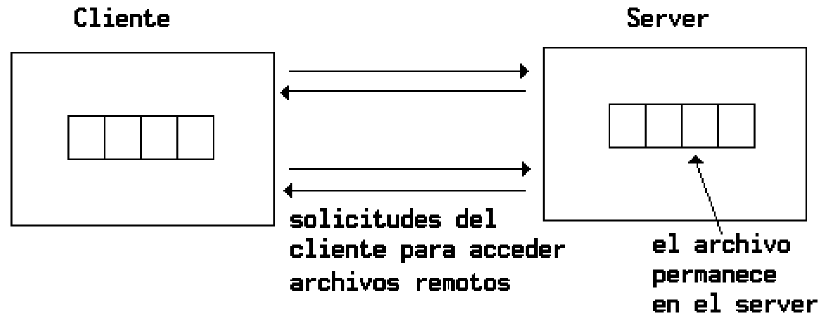


Fig. 25.2. - Modelo Remote Access.

25.4. - **EL SERVICIO DE DIRECTORIOS**

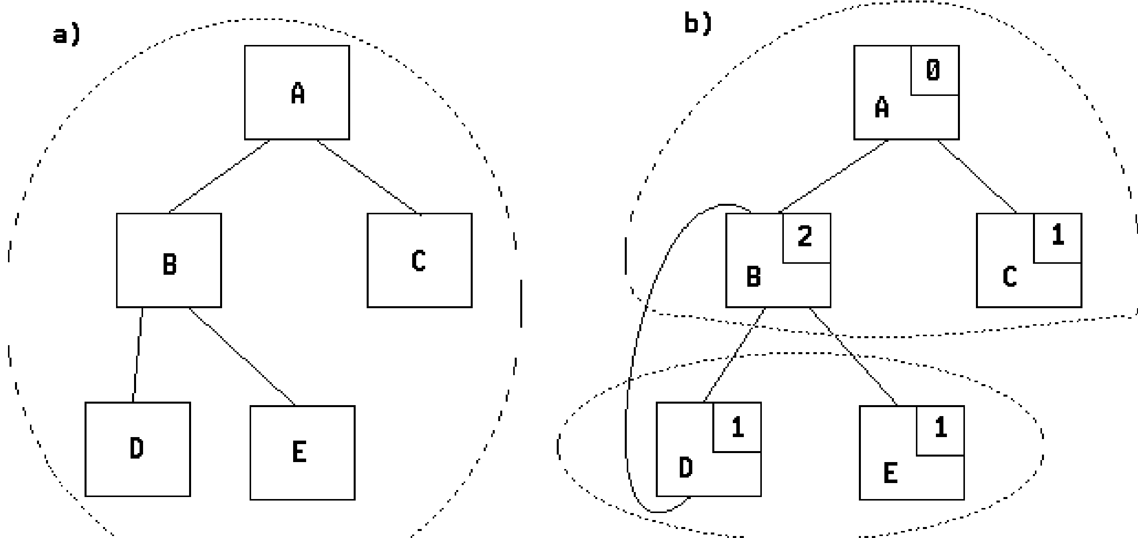
En este tipo de servicios son provistas todas las operaciones necesarias para operar a nivel de directorios y la movilidad de los archivos entre los diferentes directorios.

Para nombrar los archivos y directorios (se pueden ver como archivos a su vez con algún carácter que los identifique propiamente por ejemplo) el servicio provee un alfabeto y un conjunto de reglas (sintaxis) para definirlos.

Como es conveniente poder agrupar los archivos con características comunes (por ejemplo porque pertenecen a un usuario determinado o son archivos ejecutables de uso público), el servicio de directorios provee operaciones para la creación y borrado de directorios.

A su vez cada directorio puede contener subdirectorios dando lugar de esta manera a una **estructura jerárquica de tipo árbol de directorios**, el que se conoce como sistema jerárquico de archivos.

En algunos sistemas es posible la utilización de apuntadores a un directorio arbitrario, con lo cual es posible construir **gráficas** arbitrarias de directorios. Si bien éstas son mucho más poderosas que los árboles, acarrean



a) Directorio contenido en una máquina.

b) Grafo de directorio de dos máquinas.

Fig. 25.3.

implícitamente un peligro mayor como es el poder crear directorios huérfanos al eliminar un directorio.

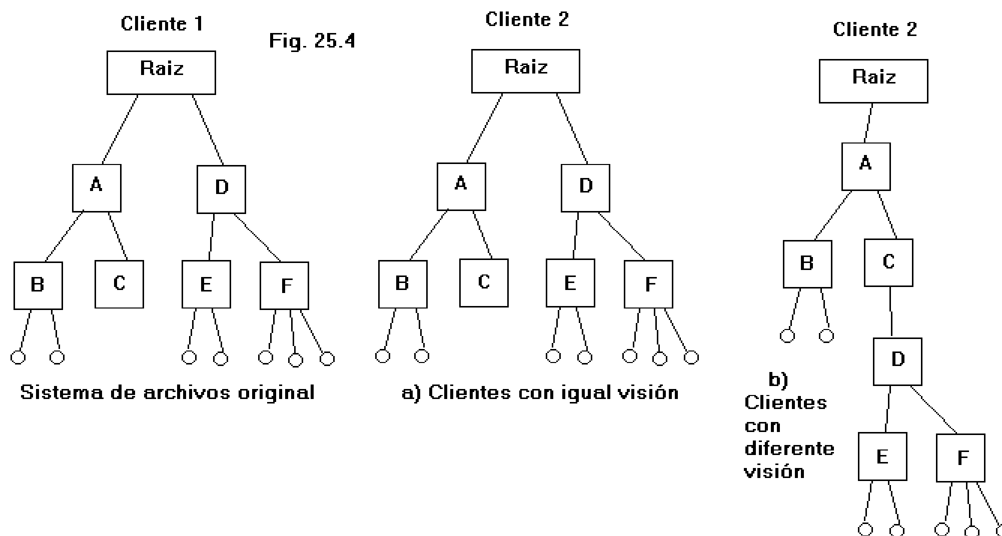
Se puede establecer que en una estructura de directorios jerárquica se puede eliminar un enlace con un directorio si el directorio apuntado está vacío. En una gráfica se permite la eliminación de un enlace mientras exista al menos otro enlace. Se utiliza para ello un contador de enlaces en cada directorio que cuenta cuántos enlaces existen que apuntan al directorio actual.

No obstante si en la gráfica de directorios de la figura 25.3 b) se elimina el directorio A, el contador de enlaces de B pasa a 1 pero los directorios B, D y E junto con sus archivos se convierten en huérfanos.

En un sistema distribuido este problema es mucho más grave y difícil de detectar y detener, ya que es casi imposible tener una visión instantánea en todo momento de lo que está sucediendo en todo el sistema (recordar el deadlock visto en los capítulos anteriores).

Otro punto a considerar y de singular importancia en el diseño de un sistema distribuido es la visión jerárquica del sistema de directorios que debe existir en todo momento; siendo las posibilidades que todos los clientes tengan la misma visión o no del file system.

El caso en que no tienen la misma visión generalmente corresponde al montaje remoto (Fig. 25.4 caso a) siendo su ventaja la flexibilidad y su implementación casi directa, en tanto que en el otro caso el file system se ve igual para todos los procesos como en un sistema de tiempo compartido y es fácil de comprender y programar (Fig. 25.4 caso b).



25.5. - **TRANSPARENCIA de los NOMBRES**

Sin embargo es importante mantener la transparencia respecto de la ubicación y la independencia respecto a la posición. Con la primera se nota el hecho de que no es posible aún conociendo el path saber la ubicación del archivo, en tanto que en la segunda se dice que un sistema es independiente de la posición si los archivos pueden desplazarse de una máquina a otra sin que cambien sus nombres.

Un sistema que incluye los nombres del servidor o la máquina en el nombre de la ruta de acceso no es independiente con respecto a la ubicación.

Tampoco un sistema con Montaje Remoto por qué no es posible mover un archivo desde un grupo (unidad de montaje) a otro y conservar el antiguo nombre de la ruta de acceso.

Como consecuencia de lo dicho anteriormente existen tres métodos de nombrar archivos y directorios en un sistema distribuido:

- Nombre máquina + path (ruta de acceso) Ej: /maquina/path/file1.
- Montaje de archivos remotos en la jerarquía local.
- Espacio de nombres con la misma apariencia en todas las máquinas.

Los dos primeros son sencillos de implementar y sirven para conectar otros file systems ya existentes que no están diseñados para uso distribuido. El tercero es más difícil pero es necesario si se quiere tener la visión de una única máquina.

25.6. - **NOMBRES de DOS NIVELES**

En cualquier sistema de computación existen dos niveles de nombres: los **simbólicos** para uso por parte de los usuarios y los **binarios** que son los nombres con los cuales el sistema realiza la referencia a los archivos. Los directorios hacen una asociación entre estos dos nombres.

En el caso de un sistema autocontenido (es decir, no tiene referencias a directorios o archivos en otros servidores) el último tipo de nombres nombrado puede ser un i-nodo local.

Las alternativas son varias:

a) El nombre binario indica el server y el nombre del archivo en ese servidor.
b) El uso de un enlace de tipo simbólico como una entrada de directorio asociada a una cadena (server, nombre del archivo). El enlace en sí es solo el nombre de una ruta de acceso.
c) Otra posibilidad es utilizar capacidades como nombres binarios. El buscar un nombre en ASCII da como resultado una capacidad que puede tener varias formas. Por ejemplo: puede tener el número físico o lógico de una máquina o la dirección en la red en el server apropiado así como un número que indique el archivo específico.

Otra cosa que sucede en un sistema distribuido y ocurre raramente en un centralizado es que al buscar un nombre ASCII se pueden obtener no uno sino varios nombres binarios (i-nodos o capacidades).

Estos pueden ser el archivo original y todos sus backups. Con estos nombres se puede empezar a buscar uno de los archivos y si no está disponible por alguna razón intentar con otro. Esto provee un cierto grado de tolerancia a fallas a través de la redundancia.

25.7. - **SEMÁNTICA de ARCHIVOS COMPARTIDOS**

Al permitir compartir archivos entre 2 o más usuarios surge el inconveniente de tener que definir en qué forma se realizarán las lecturas del archivo y las escrituras y como los cambios que realiza un usuario serán visibles o no a los otros usuarios. Esto es lo que se denomina **semántica de archivos**.

Las semánticas posibles son cuatro, a saber:

- UNIX.
- Sesión.
- Archivos inmutables.
- Transacciones.

A continuación analizaremos cada una de ellas brevemente.

25.7.1. - **Semántica UNIX**

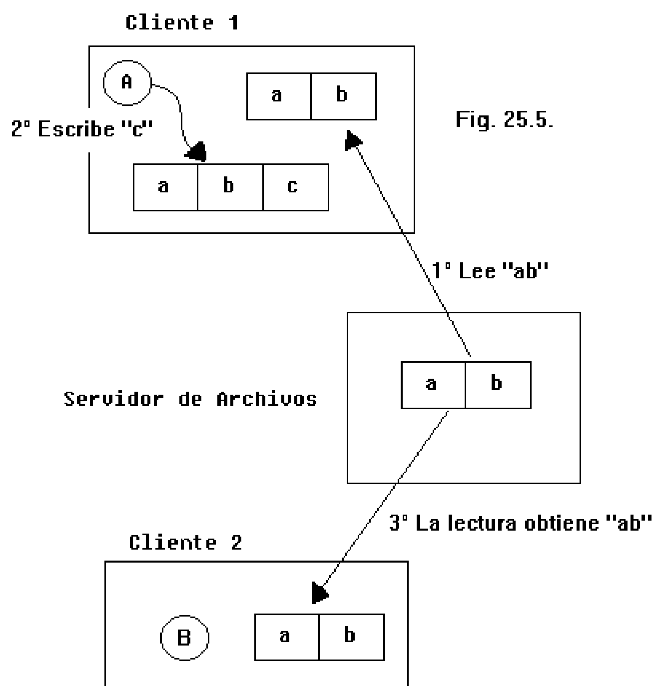


Fig. 25.5.

Establece un orden absoluto respecto al tiempo ya que secuencia las lecturas y escrituras, esto quiere decir que si se realiza un READ luego de un WRITE las modificaciones son vistas.

En el caso de un sistema donde exista un único servidor y no existen archivos ocultos por parte de los clientes, UNIX los procesa en forma secuencial. En el caso que exista ocultamiento (o sea si existe uso de caches locales en las cuales los clientes copian los archivos del servidor) entonces se obtiene un valor obsoleto en el READ luego del WRITE (ver Fig. 25.5).

Para mantener las modificaciones en los archivos se mantiene un apuntador al archivo abierto de tal manera que al realizar un WRITE se colocan los datos en el servidor.

Existe también el problema de la demora en la red debido a la cual un READ realizado luego de un WRITE llega al servidor antes que éste último.

25.7.2. - **Semántica de Sesión**

En el caso anterior si `las modificaciones se devuelven inmediatamente al servidor se logra coherencia pero el método es ineficiente, para evitar ello se 'relaja la semántica' de modo tal que las modificaciones son vistas al cerrar los archivos produciendo de esta manera la semántica de sesión.

En este caso al cerrar un archivo la versión modificada que se verá depende de la implementación. El mantenimiento de un puntero a las modificaciones realizadas en este caso no es posible ya que éstas solamente se verán al cerrar los archivos.

25.7.3. - **Semántica de archivos inmutables**

En este tipo de semántica las únicas operaciones permitidas son: CREATE y READ, por lo cual los archivos no se actualizan, sin embargo los directorios sí. Si dos procesos intentan reemplazar el mismo archivo entonces el procedimiento es el mismo que en el caso de la semántica de sesión, se procede ad-hoc dependiendo de esta manera de la implementación utilizada.

Otro problema ocurre cuando al reemplazar un archivo existe otro proceso que ya estaba leyendo el archivo anterior. Se puede permitir al lector que siga utilizando la copia anterior aunque ya no existe en el directorio. Otra solución es detectar la modificación del archivo y hacer que fallen las lecturas posteriores.

25.7.4. - **Semántica de transacciones atómicas**

En ésta se utiliza la misma idea que en las transacciones atómicas es decir vale la propiedad del todo o nada, con lo cual las modificaciones serán vistas al cierre del archivo.

25.8. - **IMPLEMENTACIÓN de un SISTEMA DISTRIBUIDO de ARCHIVOS**

25.8.1. - **Uso de archivos**

Antes de implementar cualquier sistema es útil tener una buena idea de su posible uso para garantizar la eficiencia de las operaciones. En tal sentido, Satyanarayanan realizó en el año 1981 un estudio significativo de los patrones de uso de los archivos en un sistema universitario. En éste las mediciones realizadas fueron del tipo estáticas y dinámicas. Los resultados obtenidos por él fueron más tarde verificados por Mullendor y Tanenbaum (1984) en otro estudio. En este tipo de estudios se trató de determinar diferentes características como ser tipo de acceso a los archivos (lectura, escritura), el tipo de archivos accedidos, la frecuencia, etcétera.

En este tipo de estudios es necesario tener cuidado con las mediciones obtenidas respecto a lo que ha sido medido y cómo fueron realizadas dichas mediciones ya que a priori, no existe razón alguna para asumir que estos estudios valen en otro tipo de ámbitos como puede ser el empresarial.

25.9. - **ESTRUCTURA del SISTEMA**

En el caso de un sistema distribuido ¿existe alguna característica que permite diferenciar los servers de los clientes?.

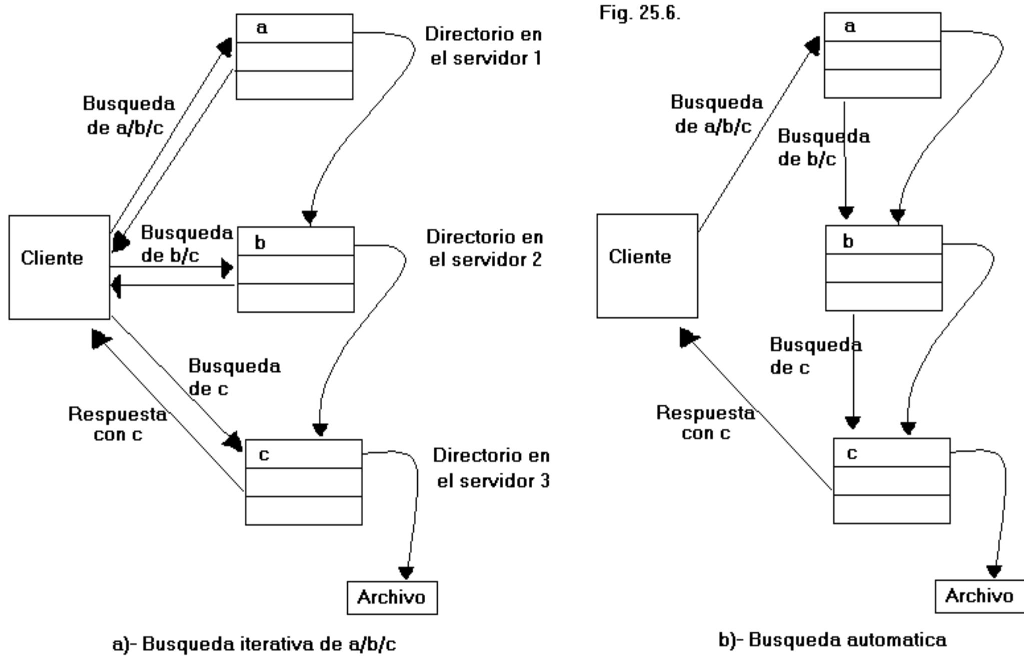
Existen sistemas en los que no existe diferencia entre los clientes y los servers y en otros sí. Sin embargo no existe razón alguna para preferir un tipo de sistema a otro.

En el caso de los primeros la máquina exporta los nombres de los directorios seleccionados de tal modo que todas las otras máquinas pueden tener acceso a ellos. También existen sistemas en los cuales el file server y el directory server son solo programas del usuario quien escoge el software a ejecutar (cliente o servidor).

Otra diferencia existente es la forma de estructurar el servicio a los archivos y a los directorios ya que es posible combinar a ambos o separarlos. En éste último caso existe más funcionalidad ya que no existe relación entre los servicios.

Por ejemplo, para abrir un archivo habrá que ir al servidor de directorios para asociar el nombre simbólico (por ejemplo, máquina + i-nodo) y después ir al servidor de archivos con el nombre binario para realizar la lectura o escritura del archivo.

Un ejemplo de esto puede ser tener un servidor de directorios en UNIX y otro servidor de directorios en DOS y sin embargo un solo servidor de archivos para el almacenamiento físico, aunque esto tiene el problema de que al haber dos servidores de directorios se necesita mayor comunicación.



Los servicios de directorios pueden estar a su vez partidos en varios servidores de directorios como se muestra en le figura 25.6. En este esquema hay dos formas de obtener la información para llegar al archivo a/b/c.

Una es que el cliente realice la consulta al primer servidor que es quien controla su directorio de trabajo y este le devuelva como respuesta un apuntador al otro servidor y al repetir la consulta al servidor b obtener el apuntador a c quien finalmente devuelve el nombre binario (caso a).

La segunda forma es más eficiente pero no puede implementarse por la RPC común ya que el que responde es un servidor distinto al que se le hizo el pedido originalmente.

Las operaciones de búsqueda son caras y por ello se trata de mejorar su desempeño por ejemplo utilizando caché de nombres recientemente utilizados para evitar la búsqueda en el directorio. Para que esto funcione es esencial que cuando el cliente utilice nombres obsoletos se le informe inmediatamente para que recurra a la búsqueda en el directorio.

Finalmente otro aspecto a considerar en la estructura es si el servidor contiene o no la información de los clientes en cuanto a las solicitudes hechas por éstos. Los métodos en estos casos se denominan **server con estado** (sí mantienen la información) o **sin estado** (en caso contrario).

En el caso de los con estado para mantener la información de los clientes es necesario que el server mantenga tablas y las actualice con las nuevas informaciones a medida que evoluciona el sistema.

En un servidor sin estado cuando el cliente envía una solicitud el servidor la satisface, envía la respuesta y elimina de sus tablas internas toda la información relativa a dicha solicitud. No guarda tampoco información alguna relativa a los clientes entre las solicitudes.

En los sin estado la solicitud debe ser autocontenida, o sea debe contener todo el nombre del archivo y el offset dentro del mismo para que el servidor pueda satisfacerle. Esto aumenta la longitud del mensaje.

Cada método tiene sus ventajas y sus desventajas sin existir mayores ventajas de un método sobre el otro.

SERVER CON ESTADO	**SERVER SIN ESTADO**
Puede existir lectura adelantada.	No necesita llamadas OPEN/CLOSE.
Es posible la cerradura de archivos.	No existe límite para el número de archivos abiertos.
Mejor desempeño.	Tolerancia de fallas.
Fácilmente aplicable la idempotencia.	No se desperdicia espacio en el server con tablas de mantenimiento.
Mensajes de solicitud más cortos.	No existen problemas al fallar un cliente.
Cuando el servidor se cae se pierde toda la información y la recuperación queda a cargo de los clientes	Mensajes de solicitud más largos
Si el cliente se cae el servidor no sabe si eliminar o no las entradas abiertas inactivas	Si el cliente se cae no hay problema para el servidor

25.9.1. - **OCULTAMIENTO (caching)**

En el modelo cliente-servidor los lugares para almacenar los archivos son:

- El disco servidor.
- La memoria del servidor.
- El disco cliente (si existe).
- La memoria cliente.

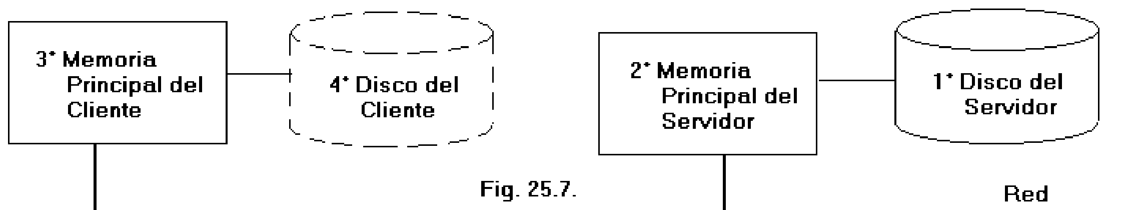


Fig. 25.7.

El disco servidor tiene la ventaja que todos los clientes ven el archivo. En contrapartida su desventaja es que existen problemas en el desempeño. Antes que el cliente pueda leer el archivo debe ser transferido del disco del servidor a la memoria del servidor y de allí a través de la red a la memoria del cliente y estas transferencias llevan cierto tiempo.

Este desempeño puede mejorarse si se utiliza la memoria del servidor en su lugar con los archivos de más reciente uso, proceso denominado **ocultamiento (caching).** Esto elimina la transferencia del archivo desde disco. No obstante es necesario algún modo para determinar qué archivos deberán permanecer en la cache.

Por otra parte, como el server puede mantener copias actualizadas de cache en su disco, entonces no existen problemas de consistencia.

El ocultamiento sin embargo puede hacerse en el cliente eliminando la transferencia desde la memoria del servidor, pero ello ocasiona problemas ya que el uso de la memoria o su disco es un problema de espacio vs. desempeño.

En general los sistemas que utilizan el disco del cliente lo hacen en memoria principal. Para ubicar su posición existen 3 opciones:

a)- Ocultar archivos en el espacio de direcciones del proceso usuario. Lo usual es que el caché sea administrado por la biblioteca de llamadas al sistema. Al operar con el archivo la biblioteca del sistema mantiene los de más uso en algún sitio para su acceso. Al terminar el proceso todos los archivos modificados si escriben en el server. Sirve si los procesos abren y cierran varias veces un archivo, pero en general no es así.

b)- Cache en el núcleo. Su desventaja es que es necesario realizar llamadas al kernel en todo momento. Su ventaja es que el cache sobrevive al finalizar el proceso. Por ejemplo un compilador de dos etapas que en su primera etapa deja el archivo listo para la linkedición no debe recurrir al servidor al necesitárselo en la segunda etapa. Si el núcleo administra la cache, entonces decide la cantidad de memoria necesaria.

c)- Colocar una cache en un proceso independiente a nivel del usuario. Su ventaja es que libera al microkernel del código del sistema de archivos, es más fácil de programar y resulta más flexible. Si el proceso administrador del cache ejecuta en una máquina con memoria virtual algunas páginas podrían ser removidas de la memoria principal del cliente perdiendo de este modo sentido el ocultamiento. Sin embargo esto se evitaría si el proceso puede fijar las páginas en memoria principal.

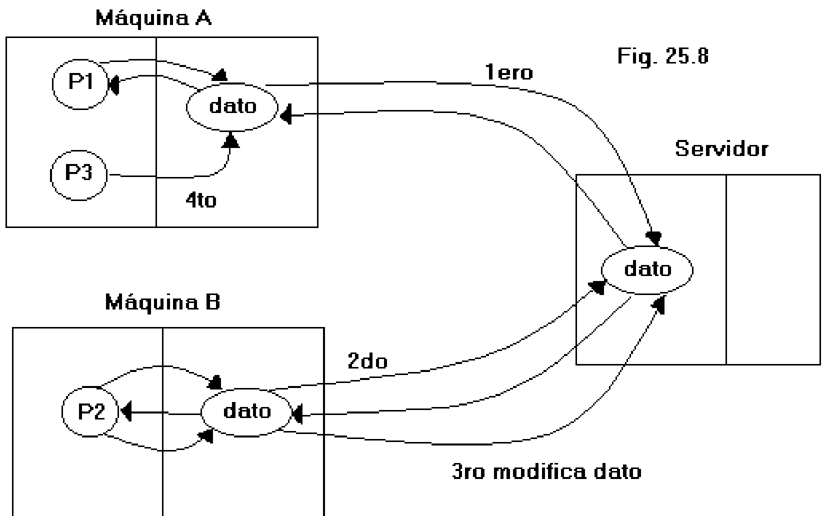
En general la cantidad de RPC's es mayor si existe ocultamiento. Sin embargo si la red de transmisión es muy rápida entonces las RPC's adicionales de las llamadas no consumirán mucho tiempo por lo que el desempeño dependerá de la tecnología existente.

25.9.2. - **CONSISTENCIA**

Ocultando a través del cliente se produce inconsistencia, por lo que la solución es adoptar la semántica apropiada.

El problema mayor se produce en la escritura del archivo. Para eliminar la inconsistencia es necesario el uso de la escritura a través del cache (write through); mediante el cual una modificación hecha en ella se envía de inmediato al server por lo que al leer el archivo desde el server se obtiene el valor más reciente.

Otro problema que se visualiza en la figura 25.8 puede ocurrir cuando desde el momento en que la máquina A obtuvo el dato otro cliente lo actualizó y cuando el proceso P3 lo solicita obtiene una versión desactualizada del mismo. Su solución puede ser que el administrador del caché en A verifique al servidor antes de entregar el archivo al cliente si el mismo está actualizado, ya sea por medio de versiones o a través de marcas temporales.



En el caso del write through para mejorar el desempeño el tráfico en la red en las escrituras es igual que en el caso de no-ocultamiento, entonces lo que se hace es realizar la escritura cada cierto intervalo de tiempo ya que leer un bloque grande es más eficiente que varios pequeños (escritura demorada). Esto obscurece la semántica ya que si otro proceso lee el archivo lo que verá dependerá de la sincronización de los eventos.

El siguiente paso es utilizar la semántica de sesión y solo escribir el archivo una vez que éste se cierra (escritura al cierre).

Otro método consiste en utilizar un algoritmo de control centralizado por el cual al abrir un archivo se le envía un mensaje al server el que mantiene un registro de los archivos abiertos y se administra su apertura según se quiera utilizar el archivo para lectura o escritura. Si un proceso quiere usar el archivo para escritura el servidor puede informar a los clientes que lo tienen en sus cachés que invaliden sus copias antes de otorgar el acceso. Al cerrar el archivo se informa al server para actualizar la información.

Método	**Comentarios**
Escritura a través del caché	Funciona pero no afecta el tráfico de escritura
Escritura demorada	Mejor desempeño pero es posible que la semántica sea ambigua
Escritura al cierre	Concuerda con la semántica de sesión
Control centralizado	Semántica UNIX, pero no es robusto y es poco escalable

Algoritmos para administrar el ocultamiento del cliente

25.10. - **IMPLEMENTACIONES DE FILE SYSTEMS**

A continuación presentaremos y analizaremos dos implementaciones de file systems utilizados en diferentes sistemas distribuidos:

NFS (Network File System)

AFS (Andrew File System)

25.11. - **NFS**

NFS es el sistema de archivos implementado por Sun Micro Systems. Este sistema es la especificación e implementación de un sistema de software para acceso a archivos remotos a través de LANs o WANs.

Una de sus grandes ventajas es que opera en un amplio rango de máquinas, sistemas operativos, entornos y arquitecturas de red.

25.11.1. - **Introducción**

NFS ve el conjunto de workstations interconectadas como un conjunto de máquinas independientes con file systems independientes. Su gran ventaja es que permite compartir los datos entre estos file systems con una gran transparencia. Esta compartición está basada en la relación cliente-servidor que existe entre las distintas máquinas y puede ser establecida entre cualquier par de ellas ya que cualquier máquina puede ser cliente y servidor.

Dado que un directorio remoto es accesible desde cualquier máquina, ésta debe realizar un montaje primero para poder accederlo y este se vuelve parte de la jerarquía de directorios en la máquina del cliente. La semántica para ello es que el directorio a montar lo hace sobre el directorio del file system local. Por lo cual una vez realizada dicha operación, el directorio montado forma parte del subárbol local reemplazando al subárbol existente hasta ese momento en ese punto de montaje. A partir de aquí los usuarios pueden acceder a los archivos remotos en forma totalmente transparente aunque es necesario realizar la operación de montaje en forma no transparente.

Para ilustrar el montaje consideremos el file system de la figura 25.9 donde tenemos tres File Systems independientes en las máquinas U, S1 y S2.

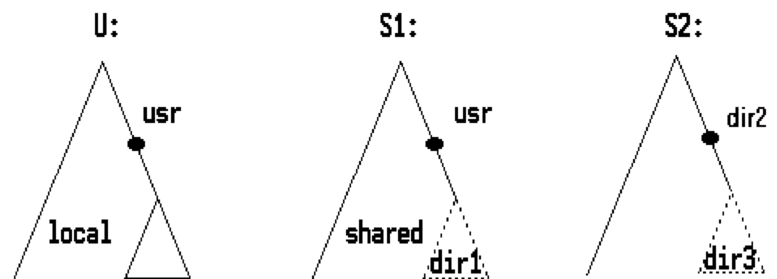


Fig. 25.9. - File Systems independientes.

En la figura 25.10 podemos ver los resultados de haber realizado el montaje de S1:/usr/d1 sobre U:/usr/local. Los usuarios de U pueden acceder a los archivos dentro de d1 utilizando el path /usr/local/d1 en U una vez completado el montaje. A su vez este directorio no es visible desde fuera.

No obstante, no existe transitividad ya que el cliente que monta un FS remoto no obtiene acceso a los FS montados sobre éste en la máquina remota.

Dos propiedades interesantes que ofrece esta implementación son:

Los montajes en cascada: es decir, montar un file system sobre otro también montado. En la Fig. 25.10 se montó sobre el directorio montado desde S1 al dir3 de S2.

La movilidad: esto es si un F.S. compartido es montado sobre los directorios home de los usuarios en todas las máquinas del sistema, un usuario puede loguearse en cualquier estación de trabajo y tener su entorno propio.

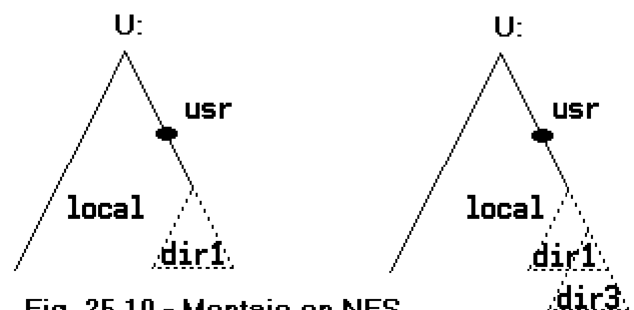


Fig. 25.10.- Montaje en NFS

La especificación de NFS distingue dos tipos de protocolos para realizar las distintas operaciones de montaje. los cuales se hallan implementados a través de primitivas de RPC's construidas sobre un protocolo de representación externa de datos (XDR).

Los protocolos son :

- Mount Protocol y
- NFS Protocol.

25.11.2.- **El Mount Protocol**

Este protocolo es utilizado para establecer la conexión lógica inicial entre el servidor y el cliente. En la implementaron de Sun cada máquina tiene un proceso servidor ejecutando fuera del kernel que hace las funciones del protocolo.

Una operación de montaje requiere el nombre del directorio remoto a ser montado y el nombre del servidor que almacena dicho directorio. Al pedido de montaje se lo relaciona con el correspondiente RPC y es enviado al servidor. Este mantiene una lista de exportaciones que especifica los file systems locales exportados para montaje y las máquinas permitidas para ello.

El servidor al recibir un pedido de montaje verifica dicha lista y en caso de tener el permiso adecuado retorna un **file handle** al cliente el cual es la unidad componente de todo pedido.

El file handle contiene toda la información que el server necesita para distinguir un archivo individual que se guarda.

En términos de UNIX, el file handle consiste de un identificador del archivo dentro del sistema y un número de i-nodo para identificar exactamente al directorio montado dentro del FS exportado.

El server mantiene también una lista de las máquinas clientes y sus correspondientes montajes para uso administrativo.

Usualmente un sistema tiene un esquema de montaje estático en tiempo de booteo el cual puede variar a posteriori.

Es importante recalcar que la operación de montaje solo afecta la visión del cliente y no la del server.

25.11.3. - **El NFS Protocol**

Este protocolo provee un conjunto de RPC's para realizar las operaciones remotas correspondientes a:

- Lectura de un conjunto de entradas de directorio.
- Lectura y escritura de archivos.
- Búsqueda de un archivo dentro de un directorio.
- Acceso a los atributos de un archivo.
- Manipulación de links y directorios.

De más esta decir que dichos procedimientos pueden ser invocados una vez obtenido el handle del directorio correspondiente.

Los servidores de NFS no mantienen la información de los clientes desde un acceso al siguiente, es decir, son sin estado.(nótese que no existen operaciones OPEN ni CLOSE).

El diseño resultante es robusto ya que no hay que tomar medidas especiales en caso de la caída del servidor.

Si bien antes mencionamos que el servidor mantiene una lista de las exportaciones hechas a los clientes lo cual es contrario a la modalidad de servidores sin estado, esto no afecta la correcta operación de NFS ya que no es necesario restaurar la lista luego de una caída del servidor.

Una implicación importante de los servidores sin estado y la sincronía de RPC's es que los datos modificados deben ser 'comiteados' antes que sean devueltos al cliente. Si bien de este modo se pierden las ventajas de tener caches locales de los clientes, en caso que el server caiga, los datos se mantendrán íntegros.

Una llamada a un procedimiento para grabar un archivo en NFS se garantiza que es atómica y no se mezcla con otras llamadas para grabar el mismo archivo. Sin embargo, el protocolo de NFS no provee mecanismos de

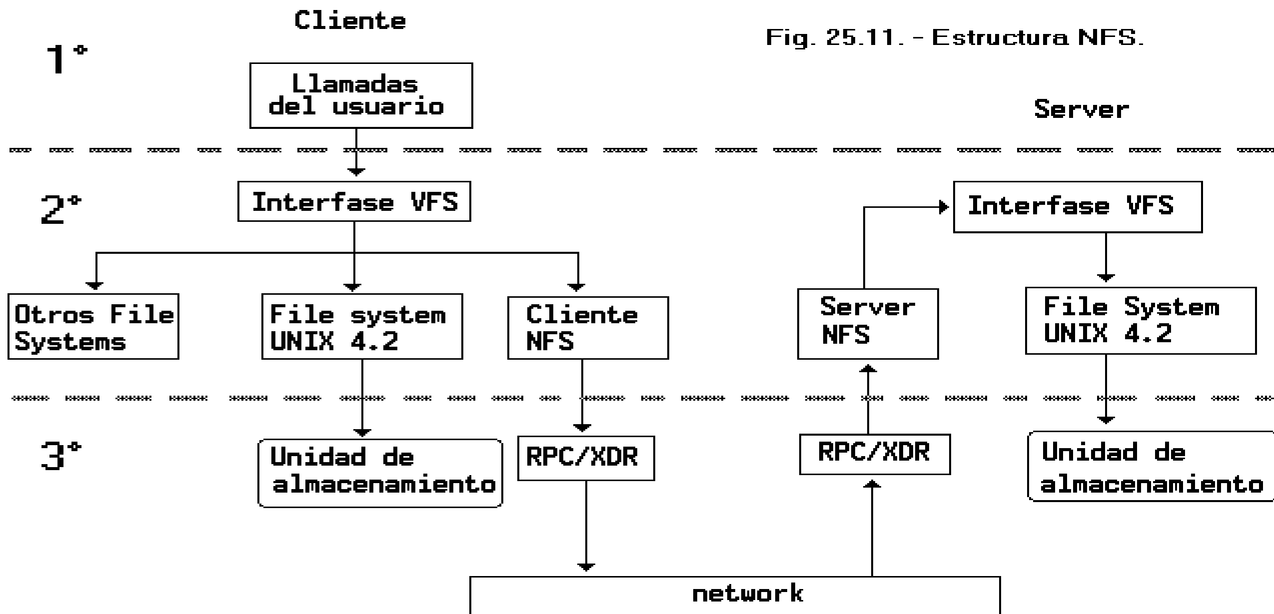


Fig. 25.11. - Estructura NFS.

control concurrentes para el caso de las escrituras de un mismo archivo, y ya que una operación de grabación puede descomponerse en varias RPC's, dos o más usuarios que graben sobre el mismo archivo en forma remota pueden obtener sus datos entremezclados.

La moraleja es que ya que el bloqueo de archivos es una operación netamente con estado se debe proveer un servicio por fuera de NFS que administre este problema (por ejemplo Sun OS lo provee).

25.11.4. - **ARQUITECTURA NFS**

Esta arquitectura que puede visualizarse en la figura 25.11 consta de 3 capas: UNIX file system, Virtual file system y NFS propiamente dicha. A continuación resumiremos las características y funciones de cada una de ellas.

25.11.4.1. - **UNIX file system**

Compuesta por las operaciones open, read, write y close y los file descriptors.

25.11.4.2. - **Virtual file system**

Define una interfase que separa las operaciones genéricas sobre el file system de su implementación. Pueden coexistir varias implementaciones en una misma máquina para permitir un acceso transparente a los distintos tipos de file systems montados localmente.

NFS se basa en una estructura para representar los archivos denominada **vnodo** el cual contiene un identificador numérico que permite identificar al archivo en forma única a lo largo de la red (recuérdese que los i-nodos de UNIX son únicos dentro de un solo file system).

El VFS distingue archivos locales de remotos y los archivos locales se diferencian de acuerdo al tipo de file system. En forma similar a UNIX, el kernel mantiene una tabla en la que guarda la información de los detalles del montaje en el que tomó parte como cliente. Más aún, los vnodos de cada directorio que fue montado se mantienen en memoria constantemente de manera tal que los requerimientos para esos directorios serán relacionados con el correspondiente file system vía la tabla de montaje. Esencialmente las estructuras de vnodo conjuntamente con las tablas de montaje proveen un puntero para cada archivo hacia el file system del que desciende así como hacia el file system sobre el cual está montado.

Las operaciones que utiliza el VFS son:

- Específicas del file system para manejar accesos locales de acuerdo al tipo de file system.
- Invocar al protocolo de NFS para accesos remotos.

25.11.4.3. - **NFS propiamente dicha**

Es la capa más baja e implementa el protocolo NFS. Se la denomina la capa de servicio NFS.

25.11.5. - **PATH-NAME TRANSLATION**

Supongamos estar en una máquina U. Podemos montar un file system realizando las correspondientes operaciones y sobre éstos realizar otro montaje.

Cada cliente tiene una visión única de su propio espacio lógico de nombres que varía según los montajes que realizó y por lo tanto se hace necesario realizar Path-Name-Translation.

El path-name-translation es 'cortar' los nombres de los file systems montados realizando una llamada (look-up) para cada par montado(nombre de componente y directorio vnodo).

Esto causa un RPC separada para cada server. Podría haber sido más eficiente manejar el path al servidor y recibir un vnodo cada vez que se encuentra un punto de montaje. Pero debido al montaje en cascada el servidor sin estado no sabría que ese montaje en el cliente está realizado sobre otro montaje anterior.

El cache del cliente guarda los nombres completos de los vnodos informados por el servidor para agilizar las operaciones.

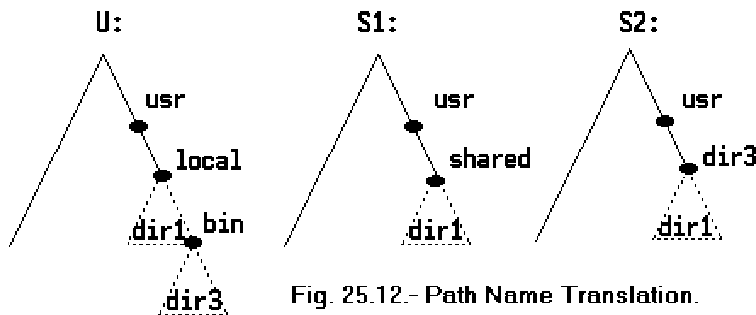


Fig. 25.12.- Path Name Translation.

25.11.6. - **OPERACIONES REMOTAS**

Las RPC's del protocolo NFS tienen su correspondencia una-a-una con las llamadas al sistema en UNIX, excepto para la apertura y cierre de archivos.

No existe correspondencia directa entre las operaciones remotas y las RPC's. En lugar de ello los bloques de archivos y sus atributos son obtenidos por las RPC's (fetch) y guardados localmente en la cache.

Los caches se dividen en **file blocks** y **file atributes** (información de i-nodo). En una apertura de archivo el kernel cliente chequea con el servidor si debe cargar los datos o si puede revalidar los atributos cargados en cache. Los bloques del archivo se usan solo si los atributos en cache son válidos.

Los atributos en caché se actualizan cada vez que llega nueva información del servidor. Se los descarta luego de 3 segundos para el caso de archivos y cada 30 segundos si son atributos de directorios.

Se utilizan técnicas de lectura anticipada y escritura demorada hasta que el server confirmó que escribió en el disco. La semántica UNIX no se preserva ya que la escritura demorada es bloqueante aún cuando el archivo haya sido abierto concurrentemente.

Es difícil caracterizar la semántica NFS. Los archivos recién creados en una máquina no serán visibles hasta tanto no transcurran 30 segundos. No se puede determinar si el grabar un archivo en un cliente será visible a otros clientes que tengan ese archivo abierto en lectura y si alguien lo abre en este momento solo verá aquellos cambios que hayan sido enviados al servidor.

Por ello NFS no posee una estricta emulación de la semántica UNIX ni de la semántica de sesión, no obstante lo cual es muy utilizada en sistemas operativos distribuidos.

25.12. - **AFS**

25.12.1. - **Introducción**

Andrew File System fue desarrollado desde 1983 en la Universidad de Carnegie-Mellon. Una de sus más formidables características es su escalabilidad ya que está pensado para soportar cerca de 5000 estaciones de trabajo (workstations).

25.12.2. - **Panorama**

AFS diferencia entre máquinas clientes (usualmente llamadas workstations) y máquinas servidores dedicados. Los clientes y servidores corren UNIX 4.2BSD y están interconectados por una LAN.

Los clientes tienen el espacio de nombres particionado en dos: local y compartido. Los servers dedicados, llamados en su conjunto VICE (nombre que proviene del software que ellos ejecutan), presentan a los clientes el espacio de nombres compartido como homogéneo, idéntico y de jerarquía de archivos de ubicación transparente. El espacio local es el root file system del cual desciende el espacio compartido.

Las workstations ejecutan el protocolo VIRTUE para comunicarse con el VICE y precisan de discos locales para guardar su espacio local. Este, si bien es pequeño, contiene la suficiente información para su auto-operación.

Mirando con un poco más de detalle los clientes y servers son estructurados en clusters interconectados por una LAN troncal (backbone LAN). Cada cluster lo forman una cantidad de workstations interconectados por una LAN y un cluster server que no es ni más ni menos que un representante del VICE. La LAN se conecta a la troncal a través de un router.

La descomposición en clusters es lo que permite salvar el problema de la escala y por ello representa uno de los puntos a favor de AFS.

La heurística básica es descargar el trabajo de los servidores hacia los clientes. Siguiendo esta idea el mecanismo seleccionado para operaciones remotas sobre archivos se basa en cargarlo completamente en la cache del cliente. No obstante si los archivos son muy grandes (por ejemplo una base de datos) y no caben en la memoria del cliente se hace necesario adicionar mecanismos extra a AFS para manejar estas situaciones.

Otras características importantes son:

Movilidad de los clientes: Los clientes pueden acceder al espacio de compartido desde cualquier workstation.

Seguridad: El VICE define una clara división ya que los programas ejecutados en sus máquinas son programas no clientes. El paquete de comunicaciones provee funciones de seguridad en la transmisión y de autenticación. La información sobre el cliente y grupos es almacenada en las bases de datos protegidas las que se hallan replicadas en cada server.

Protección: AFS provee listas de acceso para los directorios y los clásicos bits de UNIX para protección de archivos.

Heterogeneidad: A través de la definición del VICE existe una clara interfase entre diversos hardware de workstations y sus sistemas operativos. Luego algunos archivos en el directorio */bin* local son links simbólicos a archivos ejecutables específicos de cada máquina y residen en el Vice.

25.12.3. - **EL ESPACIO COMPARTIDO DE NOMBRES**

Está constituido por unidades inusualmente pequeñas denominadas **volúmenes**, a los cuales generalmente se los asocia con los archivos de un cliente y permiten la identificación y localización unívocamente de cada uno de ellos.

Los volúmenes son 'unidos' similarmente al mecanismo de montaje de UNIX; con la diferencia que en este último solo una partición entera de disco puede ser montada y en cambio una parte del disco puede tener varios volúmenes en AFS.

El VICE es asociado a un identificador de bajo nivel: el **fid**. Cada entrada de directorio en AFS mapea un componente de path-name con un fid.

Un fid tiene 96 bits de longitud y consta de tres componentes de igual longitud.: **volume number**, **vnode number** y un **uniquifier**: El vnodo se usa como un índice en un arreglo que contiene los i-nodos de los archivos en cada volumen. El uniquifier permite la reutilización de los números de vnodo.

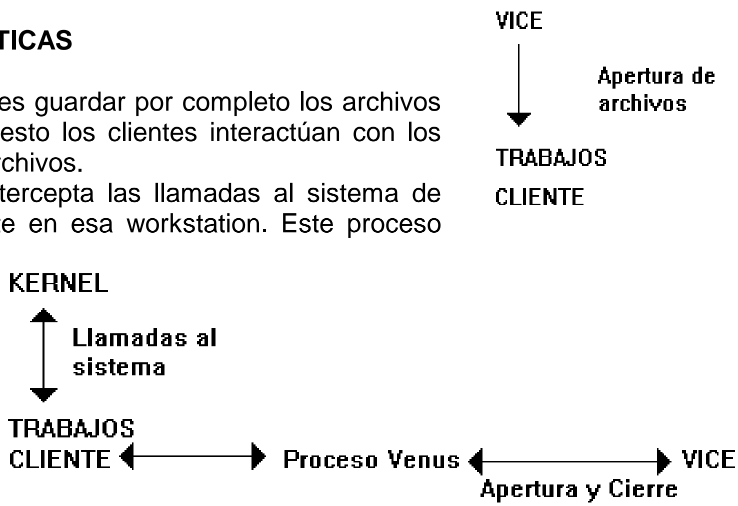
Los fid's tienen la propiedad de transparencia de ubicación, es decir al ser movidos de server a server, sus entradas siguen siendo válidas.

La información de ubicación se mantiene en una base de datos de ubicación de volúmenes replicada en cada server. De esta manera un cliente puede identificar la ubicación de cualquier volumen. Esta agregación de archivos en los volúmenes es la que hace posible el mantenimiento de las bases de datos en un tamaño manejable.

Para balancear la utilización de los servers y el espacio en disco, los volúmenes necesitan ser migrados. Esta operación se realiza en forma atómica de la siguiente forma:

> Cuando un volumen es colocado en la nueva ubicación, su server original actúa como forwarding y por lo tanto la base de datos de ubicación no necesita ser actualizada sincrónicamente. Mientras el volumen está siendo transferido, el server original aún recibe los updates los cuales son enviados más tarde al nuevo server. En algún punto el volumen es desconectado para que las nuevas modificaciones sean procesadas, luego el nuevo volumen es levantado en el nuevo server completando la operación.

25.12.4. - **OPERACIONES DE ARCHIVOS Y SEMÁNTICAS**



El principio fundamental en esta arquitectura es guardar por completo los archivos en las cachés locales a los clientes. De acuerdo a esto los clientes interactúan con los servidores VICE solo durante la apertura y cierre de archivos.

El sistema operativo en cada workstation intercepta las llamadas al sistema de archivos y las envía a un proceso a nivel del cliente en esa workstation. Este proceso llamado **Venus** hace el cache de los archivos desde el Vice cuando se los abre y guarda las modificaciones de vuelta al cerrarlos. Esta interacción Venus – Vice solo ocurre en la apertura y cierre de archivos, las lecturas o escrituras se realizan directamente en la cache y no interviene el Venus.

Venus asume que las entradas en caché son válidas a menos que se le informe lo contrario, y por ende, no contacta al Vice al abrir un archivo que ya figura en cache.

Para manejar esta situación se utiliza un mecanismo que se llama **callback**. Cuando el cliente guarda en caché un archivo o directory el servidor actualiza su información de estado. El servidor notifica al cliente antes de permitir una modificación de la información por otro cliente.

En ese caso se dice que el server elimina el callback del cliente viejo. Un cliente puede usar un archivo en cache solo si tiene callback. Si un cliente cierra un archivo luego de modificarlo todos los otros clientes que lo tienen en caché pierden sus callback y cuando vuelvan a abrir el archivo deben obtenerlo nuevamente desde el servidor.

En este mecanismo el server mantiene la información del callback propiamente dicha y el cliente la de la validación.

Cuando una workstation rebootea Venus considera las entradas en cache como posiblemente inválidas y genera un pedido de revalidación cada vez que se quiere acceder a ella por primera vez.

Si la cantidad de información de callbacks en el servidor es grande, entonces el server puede invalidar algunos.

El Venus cachea directorios y links simbólicos para path-name-translation. Cada componente en el path se obtiene (fetch) y se establece un callback para él. Las llamadas (look-up) se hacen localmente por el Venus sobre los directorios obtenidos (fetch) utilizando fids. Al final del camino del path todas las entradas del directorio y el file están en caché y cada uno tiene su callback. Las aperturas posteriores del archivo no implican tránsito en la red a menos que un callback haya sido invalidado.

La semántica utilizada es la de Sesión. Sin embargo existen excepciones, por ejemplo, las operaciones de archivos tales como cambios en la protección a nivel del directorio, las cuales son visibles inmediatamente en toda la red cuando la operación se completa.

25.12.5. - **IMPLEMENTACIÓN**

Los procesos de los clientes tienen una interfase con el kernel UNIX con el conjunto usual de llamadas al sistema. El kernel se modifica para detectar referencias al Vice y para enviar los requerimientos al nivel de procesos Venus.

Venus tiene un cache de mapeo que asocia volúmenes con servers. Si un volumen no se halla presente, entonces Venus contacta cualquier server con el que tiene conexión, requiere la información y la almacena en su propio cache. A menos que Venus ya tenga una conexión establecida con ese servidor, el establecimiento de la nueva conexión es necesario por cuestiones de autenticación y seguridad. Cuando un archivo es encontrado se crea una copia en el caché local. Venus retornará el handle al cliente, y éste abre la copia en caché.

El file system UNIX se utiliza como sistema de almacenamiento de bajo nivel para el cliente y para el server. El cache del cliente es un directorio local en el disco de la workstation. Las entradas en este directorio son los nombres de las entradas en la cache.

Venus y los procesos del server acceden a los archivos de UNIX directamente a través de los i-nodos para evitar la costosa traducción del i-nodo al path-name. Se agregan algunas llamadas al sistema debido a que la interfase del i-nodo no es visible a los procesos cliente.

Venus tiene dos caches, una para status y otra para data, a las cuales administra con el algoritmo LRU.

Cuando un archivo es removido de cache Venus notifica al server para que elimine el callback.

En el servidor hay un solo proceso a nivel cliente para brindar todos los servicios de archivos. Este proceso utiliza técnicas multithreading con threads no desalojables para atender varios pedidos concurrentemente.

Las RPC's se integran con estos threads. Existe una conexión RPC por cada cliente pero no hay un binding entre los threads y estas RPC's de manera que cualquier thread puede atender cualquier RPC.

El uso de un solo servidor multithreading permite el ocultamiento (caching) de las estructuras de datos necesarias para atender los requerimientos. Por otro lado la caída de este servidor paraliza toda su operatoria.

Referencias bibliográficas

- Amorin, C. L.; "*A Arquitetura dos Supercomputadores*", Anales del V Congreso da Sociedade Brasileira de Computaçao, 1985.

- Ancilotti, Paolo & Boari, Maurelio; "*Versione Preliminare*", Universita di Pisa, Universita de Bologna, 1987.

- Aspray, William; "*The Stored program concept*", IEEE Spectrum, Vol. 27, Nro. 9, Septiembre 90, pag. 51.

- Bishop, Peter; "*Computadoras de la 5ta. generación*", Edit. Paraninfo, 1989.

- Bogni, Carlos & Marrone, Luis; "*Computadoras: Introducción a las arquitecturas paralelas*", Edit. Da Unicamp - Campinas, (I EBAI), 1986.

- Bogni, Carlos & Marrone, Luis; "*Arquitecturas no convencionales*", Edit. Kapelusz (II EBAI), 1987.

- Boria, Jorge; "*Construcción de sistemas operativos*", Edit. Kapelusz (IV EBAI), 1989.

- Dasgupta, Subrata; "*A Hierarchical Taxonomic System for Computer Architectures*"; IEEE Computer, Vol. 23, Nro. 3, Marzo 1990, pag 64.

- Duncan, Ralph; "*A survey of parallel computer architectures*"; IEEE Computer, Vol. 23, Nro. 2, Febrero 90, pag. 5.

- *Encyclopedia of Computer Science*, First Edition, Van Nostrand Reinhold Company, (p. 722/3), 1976.

- Ferreira Magalhaes; "*Software para tempo real*", Edit. Da Unicamp - Campinas (I EBAI), 1986.

- Flynn, Michael J., Mitchell, Chad L. & Mulder Johannes M.; "*And now a case for more complex instruction sets*"; IEEE Computer; Vol. 20, Nro. 9, Septiembre 1987, pag. 71.

- Fortes, José A. B. & Wah, Benjamin W.; "*Systolic Arrays - From concept to implementation*"; IEEE Computer; Vol. 20, Nro. 7, Julio 87, pag. 12.

- Furht, Borivoje; "*A definition of complex instruction set computer (CISC) architectures* "; IEEE Computer; Vol. 20, Nro. 5, Mayo 1987. pag. 108

- Furht, Borivoje; " *A RISC architecture with Two-Size overlapping register windows*"; IEEE Micro; Vol. 8, Nro. 2, Abril 1988, pag. 67.

- Gimarc, Charles E. & Milutinovic, Veljko M.; "*A survey of RISC processors and computers of the Mid-1980s*"; IEEE Computer; Vol. 20, Nro. 9, Septiembre 1987, pag. 59.

- Hansen; "*Operating systems principles*", Edit. Prentice-Hall, 1973.

- Hayes, John P.; "*Computer architecture & organization*", Edit. McGraw-Hill, 1978

- Hwang, Kai & Briggs, Fayé A; "*Computer Architecture and Parallel Processing*", McGraw-Hill Book Company, 1985/6.

- Kline, Raymond M.; "*Digital computer design*", Edit. Prentice- Hall, 1977.

- Kung, S. Y., Lo, S. C., Jean, S. N. & Hwang, J. N.; "*Wavefront Array Processors, Concept to Implementation*", IEEE Computer, Vol 20, Nro. 7, Julio 87, pag. 18.

- Lazzerini, Beatrice; "*Effective VLSI processor architectures for HLL computers: The RISC approach* "; IEEE Micro, Vol. 9, Nro. 1, Febrero 1989, pag. 57.

- Lemonick, Michael D.; "*Superconductors, the startling breakthrough that could change the world* ", Revista TIME, Mayo 11 de 1987.

- Lilja, David J.; "*Reducing branch penalty in pipelined processors*"; IEEE Computer, Vol. 21, Nro. 7, Julio 1988, pag. 47.

- Lister, A. M.; "*Fundamentos de los sistemas operativos*", Edit. Gustavo Gili S.A., 1986.

- Lorin, H. y Deitel, H; "*Operating system*s", Edit. Addison- Wesley, 1981.

- Lucchesi, Claudio Leonardo; "*Introduçao a criptografía computacional*"; Edit. Da Unicamp-Campinas (I EBAI), 1986.

- Madnik y Donovan; "*Sistemas Operativos*", Edit. Diana, 1986.

- Maldonado, Armando; Curso de "*Redes Locales y Procesamiento Distribuido*", E.C.I. 1989, F.C.E.N. - U.B.A.

- Mano, Morris M.; "*Arquitectura de computadores*", Edit. Prentice-Hall, 1982.

- Meinadier, J. P.; "*Estructura y funcionamiento de los computadores digitales*", Edit. AC, 1973.

- Milenkovic, Milan; "*Sistemas operativos. Concepto y Diseño*", Edit. McGraw-Hill, 1988.

- Miklosko, J. y Kotov, V. E.; "*Algorithms, Software and Hardware of Parallel Computers*"; with contributions by J. Chudík, G. David, V. E. Kotov, J. Miklosko, N. N. Mirenkov, J. Ondás, I. Plander and V. A. Valkovskii; Edit. Springer-Verlag; VEDA, Publishing House of the Slovak Academy of Sciences, Bratislava; 1984.

- O'Callaghan, Patrick; Conferencia "*Seguridad de Datos*", Depto. de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 1988.

- Pedraz, Sanz y Usategui; "*Sistemas multiprocesadores*", Edit. Paraninfo, 1988.

- Peterson y Silberschatz; "*Operating systems concepts*", Edit. Addison-Wesley, 1985/6.

- del Pino, Gustavo A & Marrone Luis; "*Arquitecturas RISC*"; Edit. Kapelusz (IV EBAI), 1989.

- Rueda, Francisco; "*Sistemas Operativos*", Edit. McGraw-Hill, 1989.

- Skillicorn D. B.; "*A Taxonomy for Computer Architectures*"; IEEE Computer, Vol. 21, Nro. 11, Noviembre 1988, pag. 46.

- Tanenbaum, A.; "*Sistemas operativos. Diseño e implementación*", Edit. Prentice-Hall, 1988.

- Teller, Patricia J.; "*Translation-Lookaside buffer consistency*", IEEE Computer, Vol. 23, Nro. 6, Junio 1990, pag. 28.

- Watson, "*Sistemas de tiempo compartido*", Edit. El Ateneo, 1977