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.

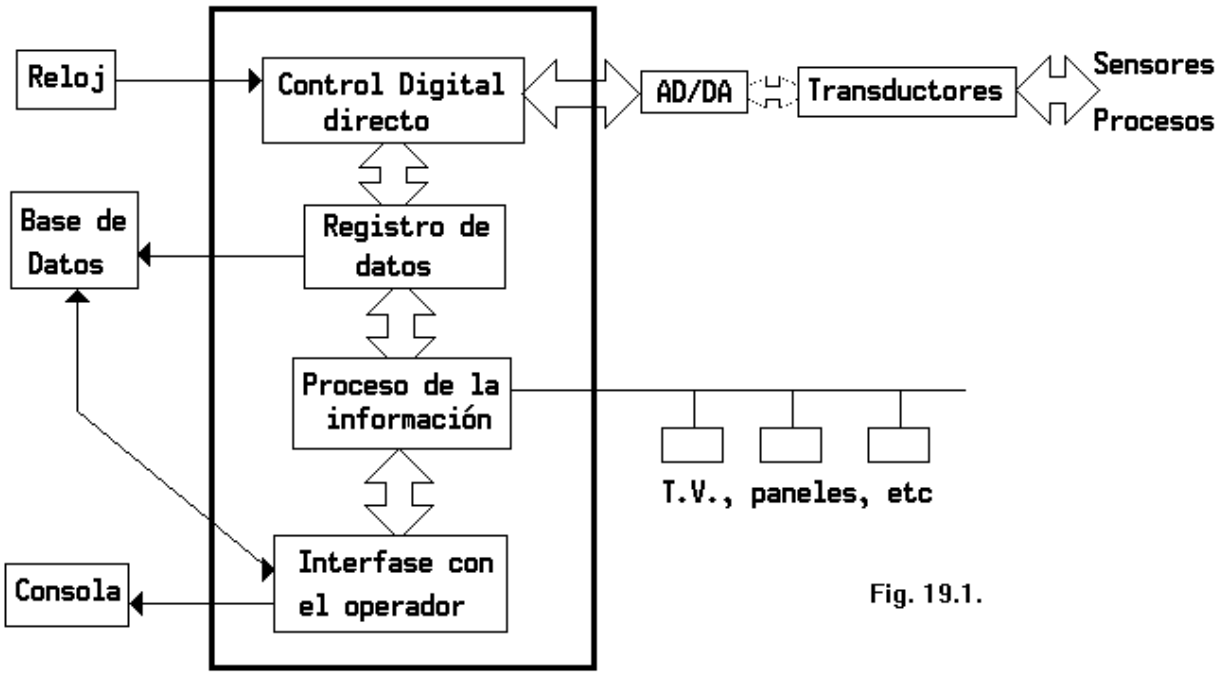

Fig. 19.1.

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.

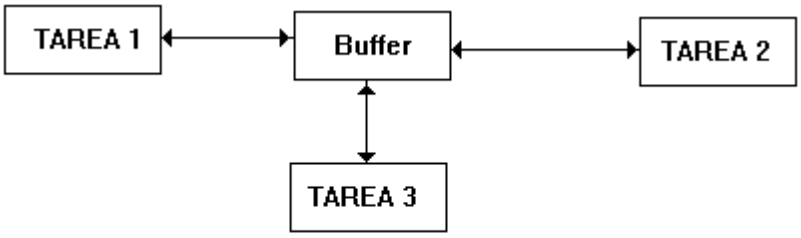

Fig. 19.2. - Comunicación a través de un área común de memoria.

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.

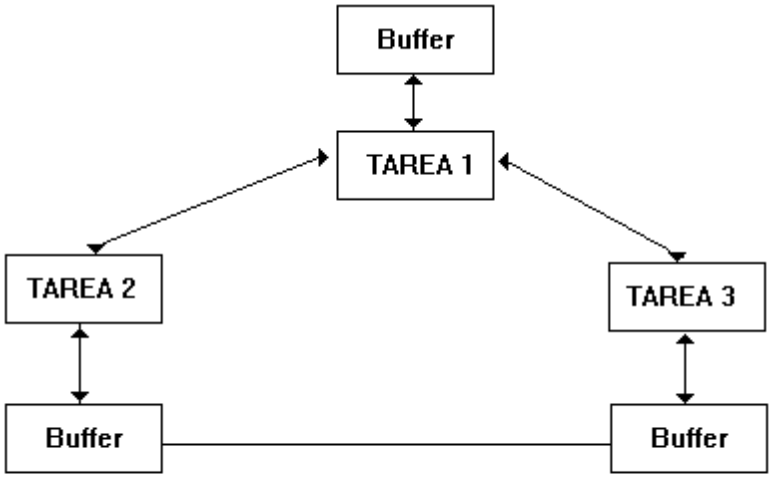

Fig. 19.3. - Comunicación mediante el intercambio de mensajes.

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.

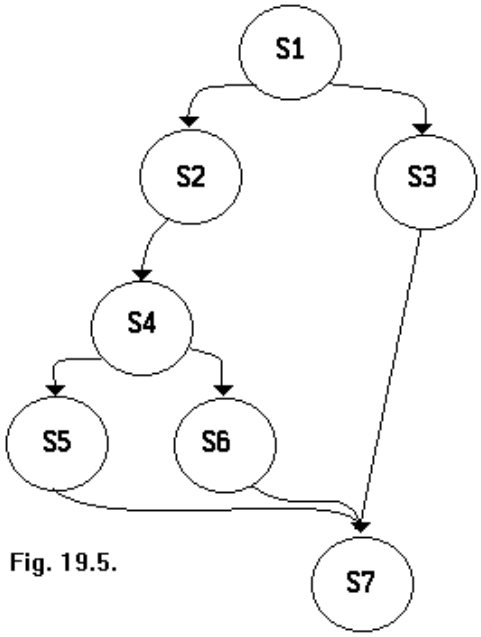

Fig. 19.5.

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);
- - - - -
```

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.

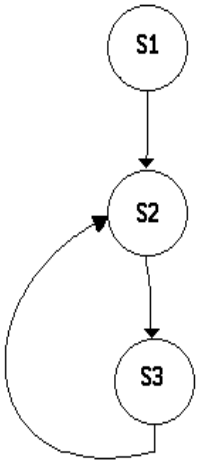

Fig. 19.6.

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)

- *DEFINICION : 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.

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).

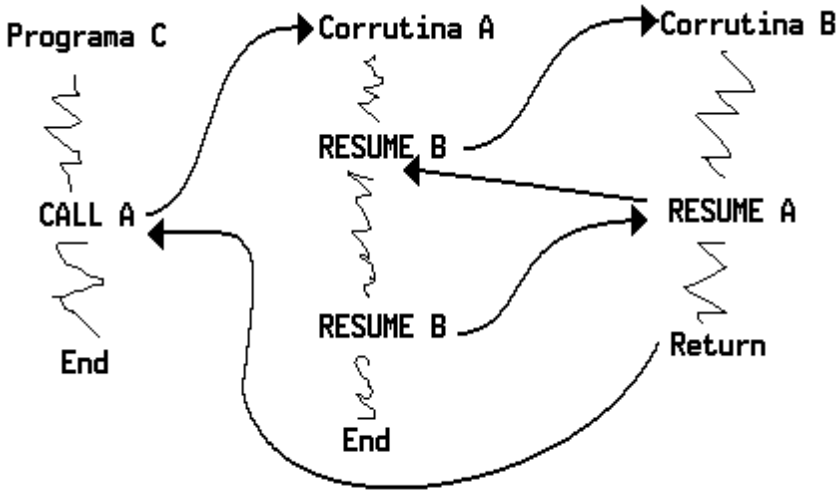

Fig. 19.7. - Esquema de funcionamiento de las Corrutinas.

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 :

```
- - -
S1;
fork L1;
S2;
```

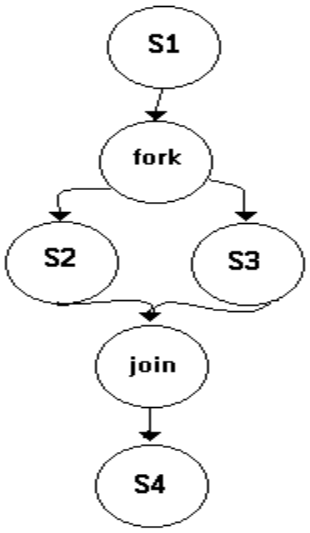

Fig. 19.8.

```
        - - -
        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.

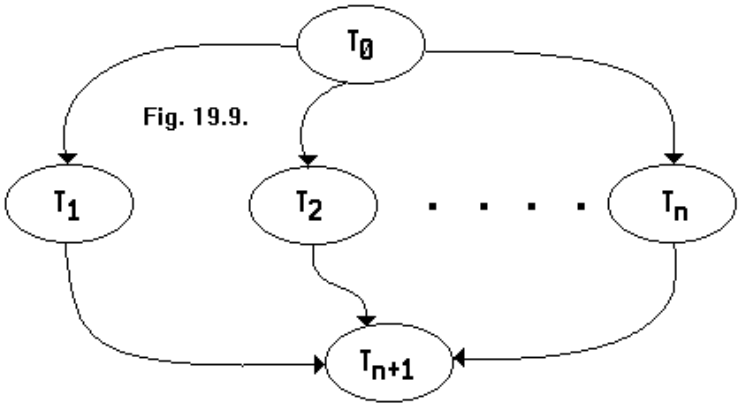

Fig. 19.9.

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
```

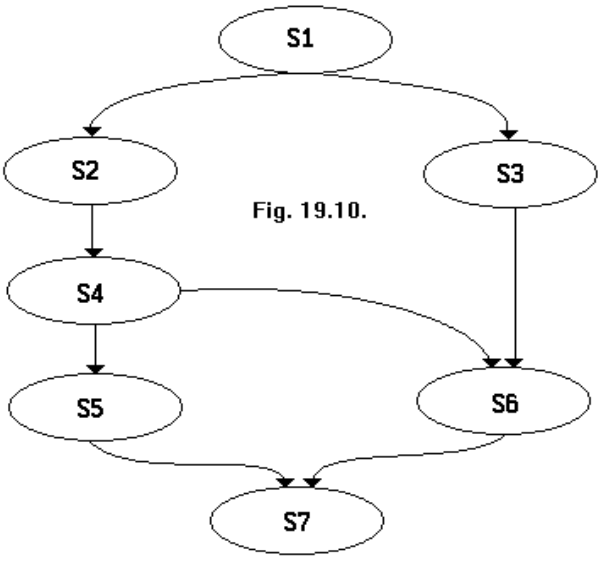

Fig. 19.10.

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.
```

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.

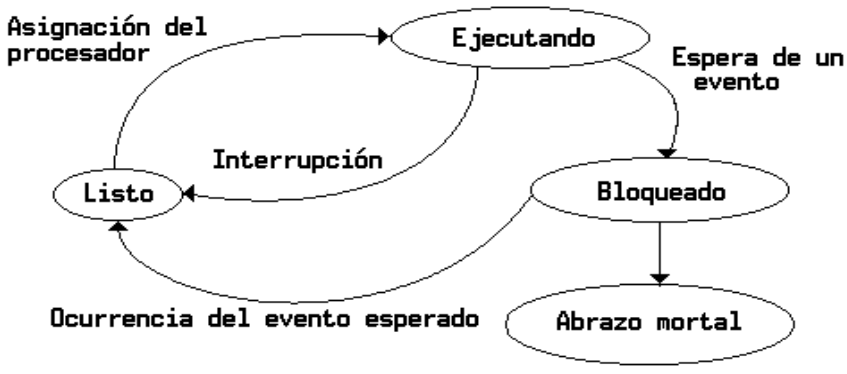

Fig. 19.11. - Diagrama de estados de un proceso.

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-

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

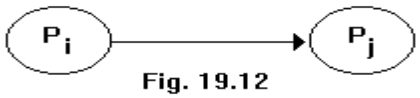

Fig. 19.12

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.

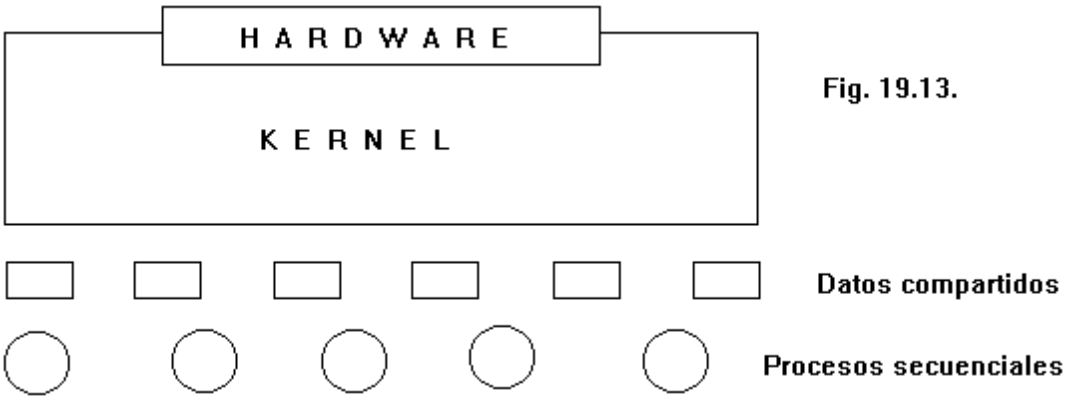

Fig. 19.13.

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).

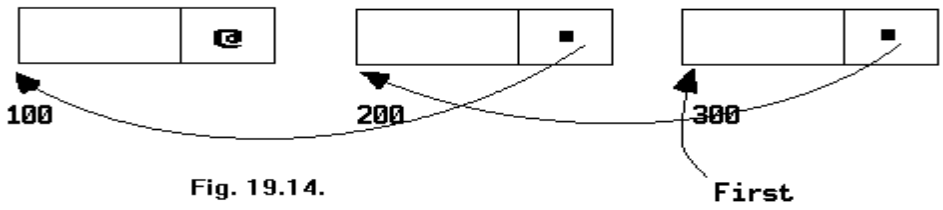

Fig. 19.14.

```
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:

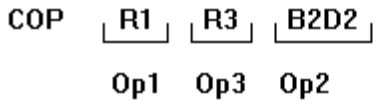

Fig. 19.16.

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 ≤ 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.

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

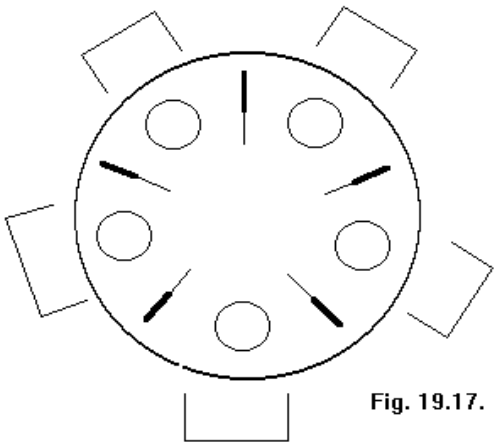

Fig. 19.17.

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 deben ser locales.

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 no 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);
```

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
```

Fig. 19.20.

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".

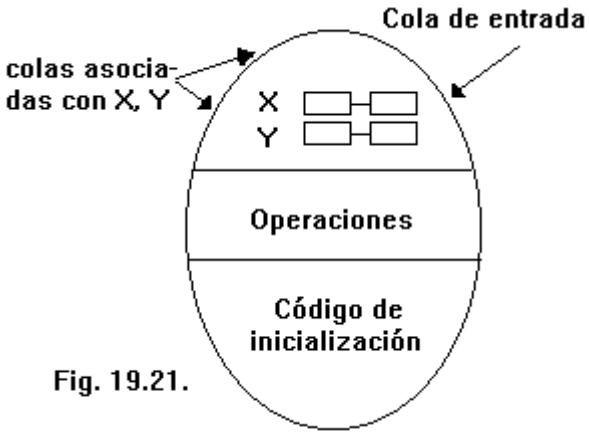

Fig. 19.21.

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);