ESPECIFICACIÓN DE EJECUCIÓN CONCURRENTE

1.4.1. ¿Qué se puede ejecutar concurrentemente?

No todas las partes se pueden ejecutar en forma concurrente. Considerando el siguiente fragmento de programa:

x:=x+1;

y:=x+2;

Está claro que la primera sentencia debe ejecutarse antes que la segunda, sin embargo si consideramos ahora:

x:=1;

y:=2;

z:=3;

Se puede observar que el orden en que se ejecuten no interviene en el resultado final. Si se dispusiera de 3 procesadores, se podría ejecutar cada una de las líneas en uno de ellos, incrementando la velocidad del sistema.

Bernstein, definió unas condiciones para determinar si dos conjuntos de instrucciones S_i y S_j se pueden ejecutar concurrentemente.

Condiciones de Bernstein:

Para poder determinar si dos conjuntos de instrucciones se pueden ejecutar de forma concurrente, se define en primer lugar los siguientes conjuntos:

- $L(s_k)=\{a_1,a_2,.....,a_n\}$, como el **conjunto de lectura** del conjunto de instrucciones S_k y que esta formado por todas las variables cuyos valores son referenciados (se leen)durante la ejecución de las instrucciones en S_k.
- $E(S_k)= \{b_1,b_2,.....,b_m\}$, como el **conjunto de escritura** del conjunto de instrucciones S_k y que esta formado por todas las variables cuyos valores son actualizados (se escriben) durante la ejecución de las instrucciones en S_k.

Para que dos conjuntos de instrucciones S_i y S_j se pueden ejecutar concurrentemente, se tiene que cumplir que:

1. $L(S_i) \cap E(S_j)=\emptyset$
2. $E(S_i) \cap L(S_j)=\emptyset$
3. $E(S_i) \cap E(S_j)=\emptyset$

Como ejemplo supongamos que tenemos :

$S_1 \longrightarrow$ a:=x + y;
$S_2 \longrightarrow$ b:=z - 1;
$S_3 \longrightarrow$ c:=a - b;
$S_4 \longrightarrow$ w:=c + 1;

Empleando las condiciones de Bernstein veremos que sentencias pueden ejecutarse de forma concurrente y cuáles no. Para ello, vamos a establecer los conjuntos de lectura y escritura correspondientes:

$L(S_1) = \{x,y\}$	$E(S_1)=\{a\}$
$L(S_2) = \{z\}$	$E(S_2)=\{b\}$
$L(S_3) = \{a,b\}$	$E(S_3)=\{c\}$
$L(S_4) = \{c\}$	$E(S_4)=\{w\}$

Luego se aplica las condiciones de Bernstein a cada par de sentencias:

Entre $S_1 \cap S_2$:

1. $L(S_1) \cap E(S_2)=\emptyset$
2. $E(S_1) \cap L(S_2)=\emptyset$
3. $E(S_1) \cap E(S_2)=\emptyset$

Entre $S_1 \cap S_3$:

1. $L(S_1) \cap E(S_3)=\emptyset$
2. $E(S_1) \cap L(S_3)=a\neq\emptyset$
3. $E(S_1) \cap E(S_3)=\emptyset$

Entre $S_1 \cap S_4$:

1. $L(S_1) \cap E(S_4)=\emptyset$
2. $E(S_1) \cap L(S_4)=\emptyset$
3. $E(S_i) \cap E(S_4)=\emptyset$

Entre $S_2 \cap S_4$:

1. $L(S_2) \cap E(S_4)=\emptyset$
2. $E(S_2) \cap L(S_4)=\emptyset$
3. $E(S_2) \cap E(S_4)=\emptyset$

Entre $S_2 \cap S_3$:

1. $L(S_2) \cap E(S_4)=\emptyset$
2. $E(S_2) \cap L(S_4)=b\neq\emptyset$
3. $E(S_2) \cap E(S_4)=\emptyset$

Entre $S_3 \cap S_4$:

1. $L(S_3) \cap E(S_4)=\emptyset$
2. $E(S_3) \cap L(S_4)=c\neq\emptyset$
3. $E(S_3) \cap E(S_4)=\emptyset$

De todo se deduce la siguiente tabla en la que puede verse que pares de sentencias pueden ejecutarse en forma concurrente**:**

	S_1	S_2	S_3	S_4
S_1	**-**	**Si**	**No**	**Si**
S_2	**-**	**-**	**No**	**Si**
S_3	**-**	**-**	**-**	**No**
S_4	**-**	**-**	**-**	**-**

Una vez identificado que se puede o no ejecutar concurrentemente se hace necesario algún tipo de notación para especificarlas-

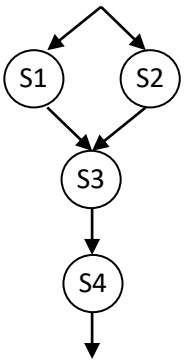
1.5.2. COMO ESPECIFICAR LA EJECUCIÓN CONCURRENTE

Veremos 2 formas de especificar la ejecución concurrente de instrucciones basadas en una notación gráficas (grafo de precedencia) y otra basada en lo que suelen utilizar diversos lenguajes de programación, el par cobegin/end.

Grafos de precedencia:

Se trata de una notación gráfica. Es un grafo dirigido acíclico. Cada nodo representará una parte (conjunto de instrucciones) de nuestro sistema. Una flecha desde A hasta B, representa que B solo puede ejecutarse cuando A haya finalizado. Si aparecen dos procesos en paralelo, querrá decir que se pueden ejecutar concurrentemente.

Para el ejemplo anterior, el grafo de precedencia sería la siguiente figura:



Sentencias COBEGIN-COEND:

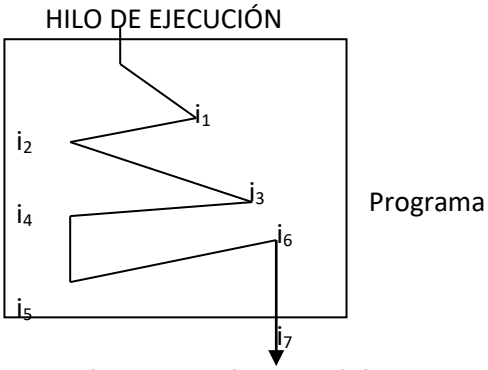
Todas las acciones que puedan ejecutarse concurrentemente las instrucciones dentro del par cobegin/end. El ejemplo anterior quedaría:

```
begin
      cobegin
            a:=x+y;
            b:=z-1;
      conend;
    c:=a-b;
    w:=c+1;
end.
```

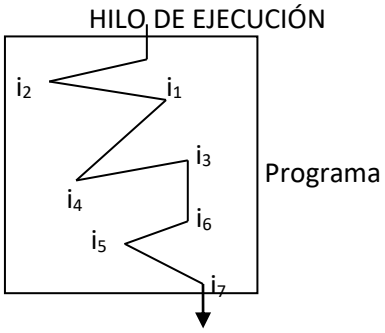
Las instrucciones dentro del par cobegin/coend pueden ejecutarse en cualquier orden, mientras que el resto se ejecuta en manera secuencial.

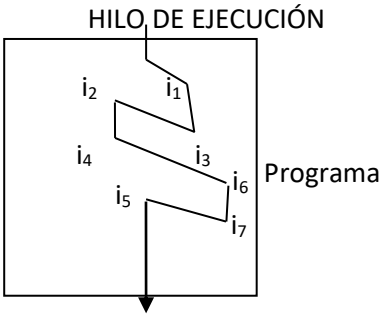
CARACTERÍSTICAS DE LOS SISTEMAS CONCURRENTES

1- **Orden de ejecución de las instrucciones**:
En los programas secuenciales, existe un orden total en la ejecución de las líneas de código. Ante un conjunto de datos de entrada

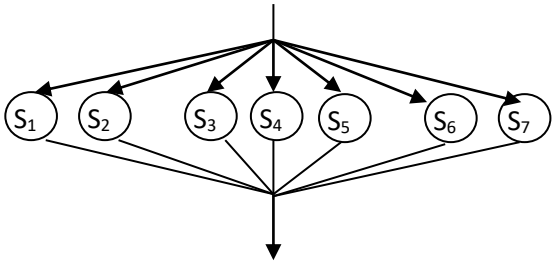


En los programas concurrentes hay un orden parcial, ante una misma entrada de datos no se puede saber cuál es el flujo de ejecución. El orden de ejecución de las instrucciones concurrentes puede variar en 2 ejecuciones distintas.





El grafo de precedencia seria:



El código con el par cobegin/coend sería:

```
Begin
    Cobegin
        i1; i2; i3; i4; i5; i6; i7,
    coend;
end.
```

2. **Indeterminismo**: el orden parcial puede provocar que se arrojen distintos resultados sobre el mismo conjunto de datos de entrada.

```
Programa incógnita
  Var x:integer;
Process P1;
  Var i:integer;
Begin
    For i:=1 to 5 do x:=x+1;
End;
Process P2;
  Var j:integer;
Begin
    For j:=1 to 5 do x:=x+1;
End;
```

```
Begin
   x:=0;
   cobegin
     P1;
     P2;
   Coend;
End;
```

¿Qué valor tendrá la variable x al terminar la ejecución?. Todo hace pensar que seria 10, pero puede ser 5, 6, 7, 8, 9, 10. Esta característica hace difícil la labor de depuración en los programas concurrentes. Intuitivamente podemos observar que el error está en el acceso incontrolado a la variable compartida.

Paradigmas de resolución de programas concurrentes

Si bien el número de aplicaciones es muy grande, en general los "patrones" de resolución concurrentes son pocos:

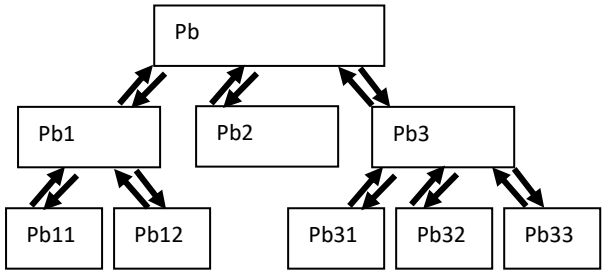
1-Paralelismo iterativo
2-Paralelismo recursivo
3-Productores y consumidores (pipelines o workflows)
4-Clientes y servidores
5-Pares que interactúan (interacting peers)

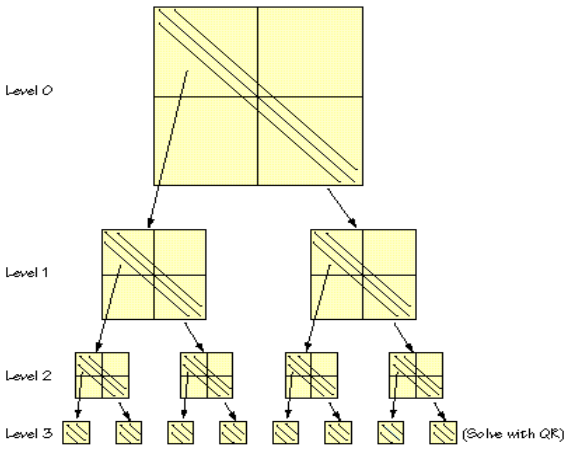
En el **paralelismo iterativo** un programa consta de un conjunto de procesos (posiblemente idénticos) c/u de los cuales tiene 1 o más loops. Luego, cada proceso es un programa iterativo.

Los procesos cooperan para resolver un único problema (x ej un sistema de ecuaciones), pueden trabajar independientemente, comunicarse y sincronizar por memoria compartida o Paso de mensaje.

Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones.
En el **paralelismo recursivo** el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (Dividir y conquistar)

Ejemplos clásicos son el sorting by merging, el cálculo de raíces en funciones continuas, problema del viajante, juegos (tipo ajedrez)

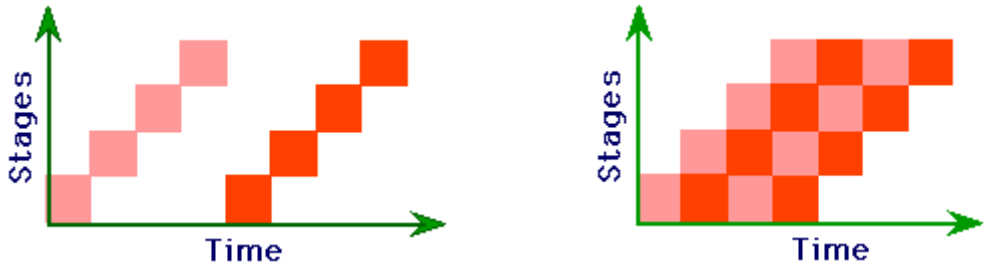




Divide y venceras

Los esquemas **productor-consumidor** muestran procesos que se comunican.
Es habitual que estos procesos se organicen en pipes a través de los cuales fluye la información.
Cada proceso en el pipe es un filtro que consume la salida de su proceso predecesor y produce una salida para el proceso siguiente.
Ejemplos a distintos niveles de SO.

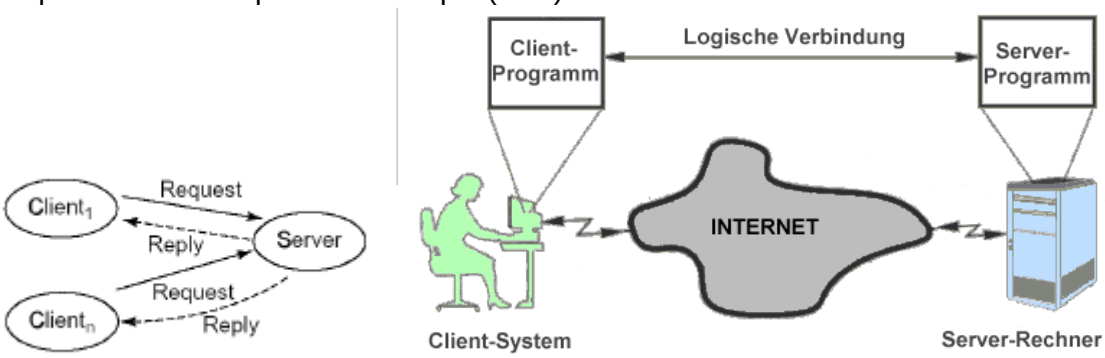
Pipe



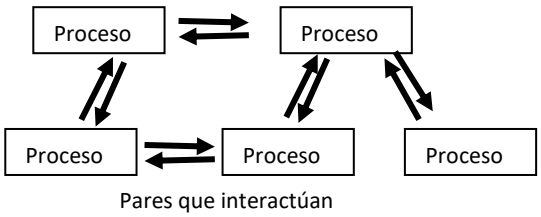
Cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido.

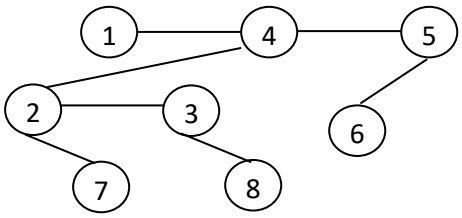
Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Unos y otros pueden ejecutarse en procesadores diferentes. Comunicación bidireccional. Atención de a un cliente o con multithreading a varios.

Mecanismos de invocación variados (rendezvous y RPC x ej en MD, monitores x ej en MC)
.
El soporte distribuido puede ser simple (LAN) o extendido a la WEB.



En los **esquemas de pares que interactúan** los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea y completar el objetivo.
Permite mayor grado de asincronismo que C/S
Configuraciones posibles: grilla, pipe circular, uno a uno, arbitraria





PROBLEMAS INHERENTES A LA EXCLUSIÓN MUTUA:

Los problemas que surgen son: el problema de exclusión mutua y el de condición de sincronización.

Lo que realmente se ejecuta concurrentemente son las líneas generadas por el compilador. Si consideramos que una instrucción como x:=x+1 da lugar a 3 instrucciones de un lenguaje ensamblador, tendríamos:

Cargar desde memoria el valor de x en un registro (LOAD x R1).

Incrementar el valor del registro (ADD R1 1)

Almacenar el contenido del registro en la posición de memoria de x (STORE R1 X)

Lo que realmente se va ejecutar en forma concurrente dentro del bucle es:

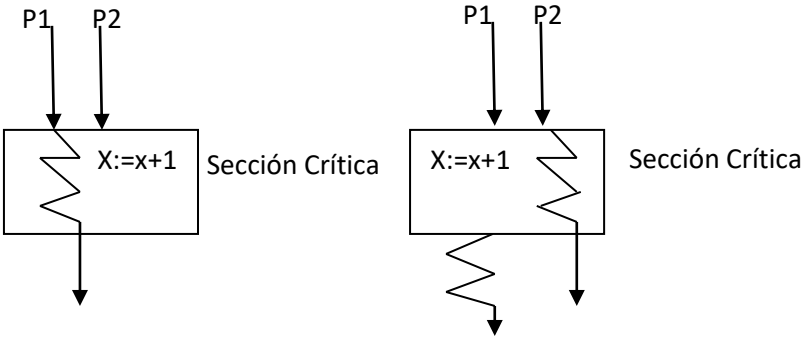
```
P1                          P2
    (1) LOAD  X   R1        LOAD   X   R1
    (2) ADD   R1  1         ADD    R1  1
    (3) STORE   R1 X        STORE    R1 X
```

Cada una de estas acciones son atómicas e indivisible, es decir se ejecutan en un ciclo de reloj del procesador cualquier intercalación de instrucciones es válido. En la siguiente figura puede observarse una traza para un intercalado para estas líneas de código, en el que el valor final de la variable x no es el esperado.

X	0	0	0	0	1	1	1
P1	1	2			3		
P2			1	2		3	

Se perdió un incremento. Si tenemos en cuenta que estas líneas de código están dentro de un bucle, podremos entender porque son posibles resultados menores a 10. Todo dependerá del número de incrementos perdidos, que en cada ejecución puede ser distintos o no producirse.

El hecho que P1 y P2 no puedan ejecutarse concurrentemente viene determinado por condiciones de Bernstein pues sus conjuntos de escritura no son disjuntos. Es decir, 2 procesos distintos están accediendo al mismo tiempo a una variable compartida para actualizarla.



A la porción de código que queremos que se ejecute en forma indivisible se denomina **sección crítica,** y queremos que se ejecuten en **exclusión mutua,** es decir solo uno de los procesos debe estar en sección crítica en un instante dado. Una vez que un proceso ha salido de sección crítica, el otro proceso puede entrar y de esta forma seguir ejecutándose los 2 procesos de manera concurrente.
Por lo cual la programación concurrente deberá ofrecernos mecanismos para especificar que partes del código han de ejecutarse en exclusión mutua con otras partes

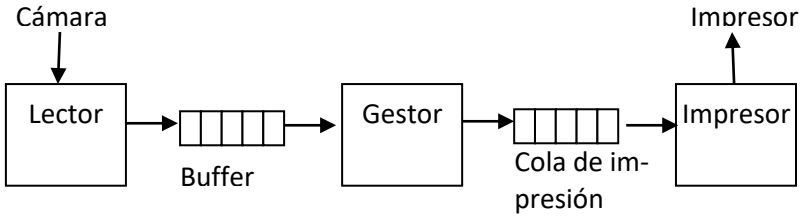
Condición de sincronización:

Para ilustrar el problema de sincronización supongamos un sistema en que existen tres procesos cuyo comportamiento es el siguiente:

Lector, que va almacenando en un buffer las imágenes capturadas desde una cámara,

Gestor, que va recogiendo las imágenes desde el buffer, las trata y las va colocando en una cola de impresión.

Impresor, que va imprimiendo las imágenes colocadas en la cola de impresión.



Suponiendo que estos procesos se ejecutan concurrentemente en un bucle infinito como se muestra a continuación:

```
Process lector;
    Repeat
        Captura imagen
        Almacena en buffer;
    Forever
end;
```

```
Process lector;
Begin
    Repeat
        Captura imagen
        Almacena en buffer;
    Forever
end;
```

```
Process gestor;
Begin
        Repeat
        Toma imagen de buffer;
        Trata imagen;
        Almacena imagen en cola;
    Forever
End;
```

Podemos observar que la solución está incompleta. Se tendría que poder responder las siguientes preguntas:
¿Qué ocurre cuando el proceso o el proceso gestor tratan de poner una imagen y el buffer o la cola están llenos?
¿Qué ocurre cuando el proceso gestor o el proceso impresor tratan de tomar una imagen y el buffer o la cola están vacíos?
Hay situaciones en las que un recurso es compartido por varios procesos, como puede ser el buffer o la cola de impresión en nuestro ejemplo, se encuentra en un estado en el proceso no puede hacer una determinada acción con él hasta que no cambie su estado. A esto se le denomina **condición de sincronización.**
La programación concurrente ha de proporcionarnos mecanismos para bloquear procesos que no puedan hacer algo en un momento determinado a la espera de algún evento (que el buffer deje de estar vacío), pero que permita desbloquearlo cuando ello haya ocurrido.

Corrección de programas concurrentes

Para que un programa concurrente sea correcto debe cumplir con las especificaciones funcionales debe satisfacer una serie de propiedades inherentes a la concurrencia. Estas propiedades se pueden agrupar en:

- **Propiedades de seguridad:** son aquellas que aseguran que nada malo va a pasar durante la ejecución del programa.
- **Propiedades de viveza:** aquellas que aseguran que algo bueno va a pasar durante la ejecución del programa.

Para entender estas propiedades, vamos a considerar el juego del pañuelo: tenemos dos equipos A y B y un juez con un pañuelo. Cada jugador de un equipo tiene un nro del 1 al 3. No puede haber dos jugadores con el mismo nro en el equipo. El juez dice un número y los rivales con el mismo numero corren para tomar el pañuelo. El jugador con el pañuelo debe volver a su lugar sin que el rival le toque la espalda.

Propiedades de seguridad:

-Exclusión mutua: hay recursos en el sistema que deben ser accedidos en exclusión mutua. Cuando esto ocurre hay q garantizar que si un proceso adquiere el recurso, los otros procesos deberán esperar a que sea liberado, para evitar resultados imprevistos (ejemplo: la toma del pañuelo, lo toma un solo jugador, si lo toman los 2 puede romperse) llevando a un mal funcionamiento del sistema.

-Condición de sincronización: hay situaciones en las que un proceso debe esperar la ocurrencia de un evento para seguir ejecutándose. => se debe garantizar que el proceso no prosiga hasta que no se produzca el evento. (ejemplo: los jugadores deben esperar hasta que digan el nro).

-Interbloqueo: cuando los procesos esperan que ocurra un evento que no se producirá, hay que garantizar que no va a ocurrir esta situación. (ejemplo: que un jugador tome el pañuelo lo guarde y se vaya a su casa, el juez esperaría el pañuelo y los jugadores q vuelva a decir un nro). También se conoce como interbloqueo pasivo o deadlock o abrazo mortal.

Problemas de vivacidad:

-Interbloqueo activo: se produce cuando el sistema ejecuta una serie de instrucciones sin hacer un proceso. (ejemplo: que dos jugadores amaguen tomar el pañuelo y no lo hacen). Es complicado la detección,, se conoce también como livelock. Nosotros nos vamos a referir a Interbloqueo pasivo.

-**Inanición**: se produce cuando el sistema en su conjunto hace procesos, pero alguno nunca progresan, pues no se les otorga tiempo de procesador para avanzar. (en ejemplo: si el juez nunca dice un número correcto de jugador). Hay que garantizar una equidad en el trato a los proceso a no ser que las especificaciones del sistema digan lo contrario. Es difícil de detectar. También se conoce como starvation.