UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos: Tema 1. Introducción.

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2025-26 (archivo generado el 8 de septiembre de 2025)

Grado en Ingeniería Informática
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Tema 1. Introducción.
Índice.

Sección 1.
Conceptos básicos y motivación.

1.1. Conceptos básicos relacionados con la concurrencia
1.2. Motivación de la Programación concurrente

Subsección 1.1.
Conceptos básicos relacionados con la concurrencia.

Concurrencia: programa y programación concurrentes

- **Programa secuencial:** Declaraciones de datos + Conjunto de instrucciones sobre dichos datos que se deben ejecutar en secuencia.
- **Programa concurrente:** Conjunto de programas secuenciales ordinarios que se pueden ejecutar *lógicamente* en paralelo.
- **Proceso:** Ejecución de un programa secuencial.
- **Concurrencia:** Describe el potencial para ejecución paralela, es decir, el solapamiento real o virtual de varias actividades en el tiempo.
- **Programación Concurrente (PC):** Conjunto de notaciones y técnicas de programación usadas para expresar paralelismo potencial y resolver problemas de sincronización y comunicación.
- La PC es independiente de la implementación del paralelismo. Es una abstracción

Programación paralela, distribuida y de tiempo real

- **Programación paralela:** Su principal objetivo es acelerar la resolución de problemas concretos mediante el aprovechamiento de la capacidad de procesamiento en paralelo del hardware disponible.
- **Programación distribuida:** Su principal objetivo es hacer que varios componentes software localizados en diferentes ordenadores (o, en general, sin memoria compartida) trabajen colaborativamente.
- **Programación de tiempo real:** Se centra en la programación de sistemas que están funcionando continuamente, recibiendo entradas y enviando salidas a/desde componentes hardware (*sistemas reactivos*), en los que se trabaja con restricciones muy estrictas en cuanto a la respuesta temporal (*sistemas de tiempo real*).

Subsección 1.2.
Motivación de la Programación concurrente.

Beneficios de la Programación concurrente

La programación concurrente es más compleja que la programación secuencial, entonces nos preguntamos:

¿ Por qué es necesario conocer la Programación Concurrente ?

Básicamente hay dos motivos para el desarrollo de la programación concurrente:

- Mejora de la **eficiencia**
- Mejoras en la **calidad**

Veremos ambos aspectos.

Beneficios P.C.: Mejora de la eficiencia.

La PC permite aprovechar mejor los recursos hardware existentes.

- **En sistemas con un solo procesador:**
  - Al tener varias tareas, cuando la tarea que tiene el control del procesador necesita realizar una E/S cede el control a otra, evitando la espera ociosa del procesador.
  - También permite que varios usuarios usen el sistema de forma interactiva (actuales sistemas operativos multisusuario).
- **En sistemas con varios procesadores:**
  - Es posible repartir las tareas entre los procesadores, reduciendo el tiempo de ejecución.
  - Fundamental para acelerar complejos cómputos numéricos.

Beneficios P.C.: Mejora de la calidad

Muchos programas se entienden mejor en términos de varios procesos secuenciales ejecutándose concurrentemente que como un único programa secuencial.

Ejemplos:

- **Servidor web para reserva de vuelos:** Es más natural, considerar cada petición de usuario como un proceso e implementar políticas para evitar situaciones conflictivas (permitir superar el límite de reservas en un vuelo).
- **Simulador del comportamiento de una gasolinera:** Es más sencillo considerar los surtidores, clientes, vehículos y empleados como procesos que cambian de estado al participar en diversas actividades comunes, que considerarlos como entidades dentro de un único programa secuencial.

Programas de cálculo intensivo

En la actualidad hay múltiples aplicaciones de cálculo intensivo que se benefician de la programación concurrente y distribuida.

- **Cálculo intensivo en múltiples CPUs**: en la actualidad, los ordenadores incorporan múltiples CPUs. Un programa secuencial puede usar únicamente una de ellas, mientras que un programa concurrente puede usar varias a la vez de forma coordinada.
- **Cálculo intensivo en GPUs**: uso de programación concurrente para cálculos intensivos en GPUs aprovechando su hardware que incorpora cientos o miles de unidades de cálculo flotante.
- **Sistemas distribuidos de cálculo**: permiten cálculos intensivos en varios ordenadores conectados a un red, con paso de mensajes entre ellos.

Aplicaciones del cálculo intensivo

Como consecuencia, la programación concurrente se ha convertido en una herramienta esencial en diversas áreas que requieren cálculo intensivo, podemos citar algunas:

- **Videojuegos, realidad virtual y aumentada:** la PC se usa para visualización, simulación física e I.A. de personajes.
- **Inteligencia artificial:** entrenamiento de redes neuronales o en general aprendizaje de sistemas de IA. También su uso una vez entrenadas.
- **Animación y *rendering*:** generación *off-line* de animaciones y efectos especiales en películas.
- **Simulación de sistemas físicos:** predicción meteorológica, cálculo de estructuras de edificios u objetos, simulación de viento o fluidos, etc...
- **Cálculo de estructuras moleculares:** simulación computacional del comportamiento y estructura de moléculas y proteínas.

Programación Web o en Internet

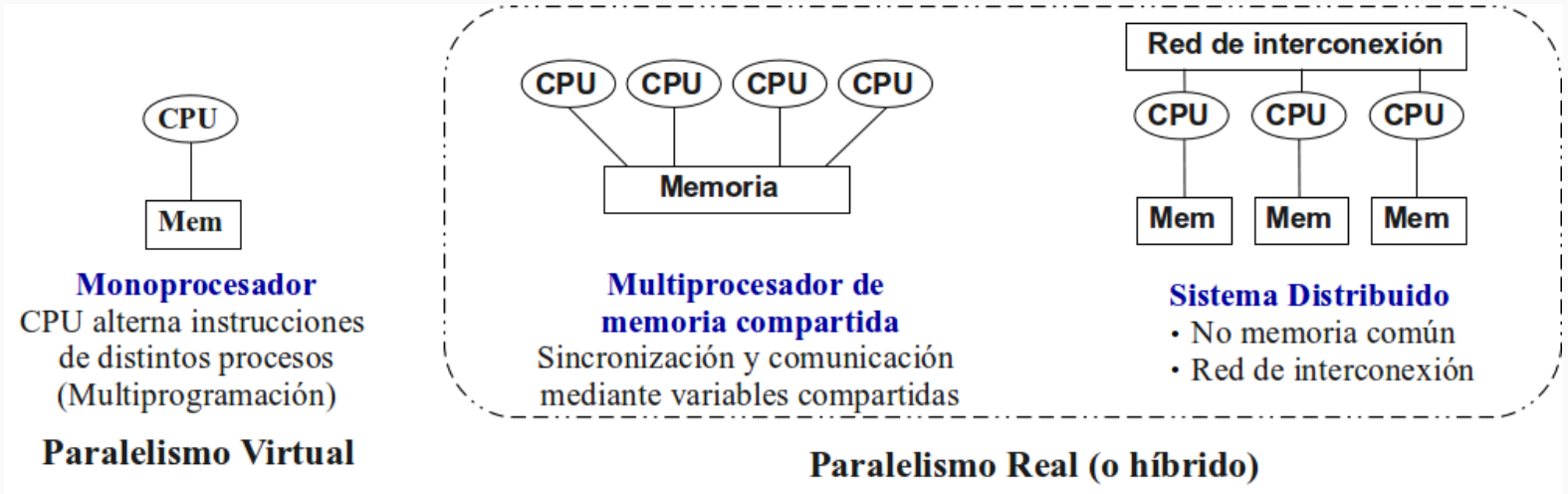
La programación distribuida constituye el modelo subyacente a la programación de aplicaciones Web, en las cuales distintos procesos (probablemente en distintos ordenadores) se comunican mediante mecanismos de paso de mensajes. Ejemplos:

- Comunicación distribuida (típicamente siguiendo el modelo *cliente-servidor*, que veremos más adelante) mediante distintas modalidades de paso de mensajes.
- Programación asíncrona en clientes Web: permite simultanear el procesamiento, las transferencias de datos y la interacción con el usuario.
- Uso concurrente de varias CPUs en un cliente Web, con coordinación y comunicación mediante paso de mensajes.

Sección 2.
Modelo abstracto y consideraciones sobre el hardware.

Subsección 2.1.
Consideraciones sobre el hardware.

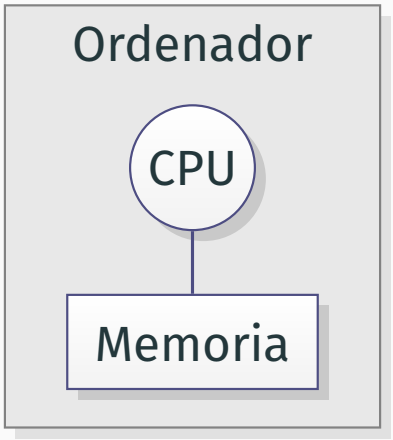
Modelos de arquitecturas para programación concurrente



Mecanismos de implementación de la concurrencia

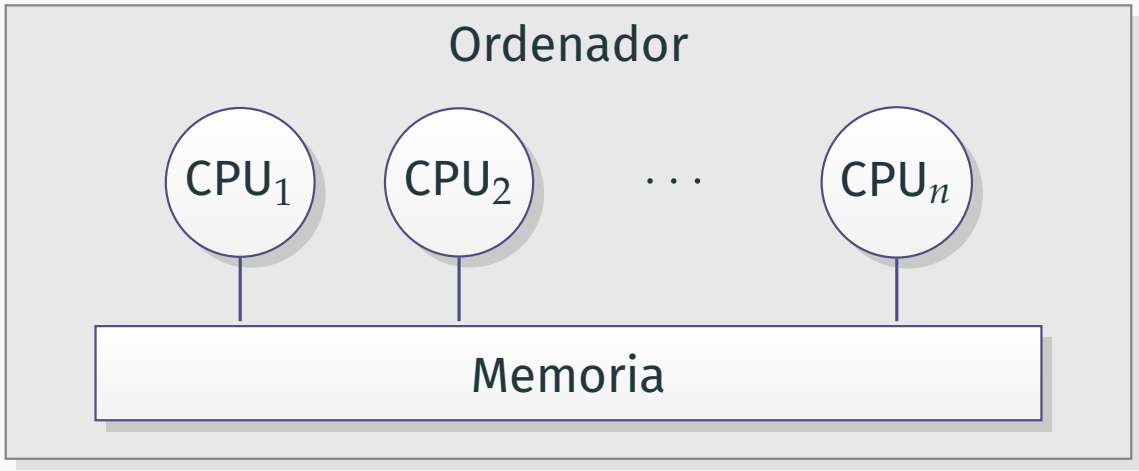
- Dependen fuertemente de la arquitectura.
- Consideran una *máquina virtual* que representa un sistema (multiprocesador o sistema distribuido), proporcionando una base homogénea para la ejecución de procesos concurrentes.
- El tipo de paralelismo afecta a la eficiencia, pero no a la corrección.

Concurrencia en sistemas monoprocesador



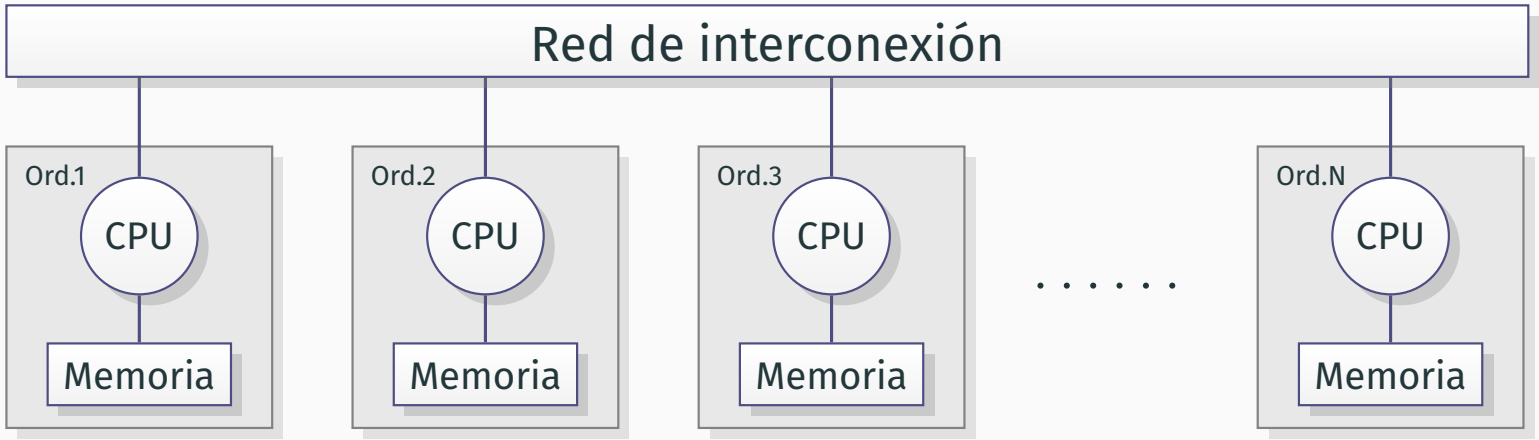
- **Multiprogramación:** El sistema operativo gestiona cómo múltiples procesos se reparten los ciclos de la (única) CPU.
- Mejor aprovechamiento CPU.
- Servicio interactivo a varios usuarios.
- Permite usar soluciones de diseño concurrentes.
- Se hace sincronización y comunicación mediante variables compartidas en la memoria.

Concurrencia en multiprocesadores de memoria compartida



- Los procesadores pueden compartir o no físicamente la misma memoria, pero siempre comparten un espacio de direcciones compartido.
- La interacción entre los procesos se puede implementar mediante variables alojadas en direccciones del espacio compartido (variables compartidas).
- Ejemplo: PCs con procesadores *multicore*.

Concurrencia en sistemas distribuidos



- No existe una memoria común: cada procesador tiene su espacio de direcciones privado.
- Los procesadores interaccionan transfiriéndose datos a través de una red de interconexión (paso de mensajes).
- **Programación distribuida**: además de la concurrencia, trata con otros problemas como el tratamiento de los fallos, transparencia, heterogeneidad, etc.
- Ejemplos: Clusters de ordenadores, internet, intranet.

Subsección 2.2.
Modelo abstracto de un proceso secuencial.

Programa secuencial.

Un **programa secuencial** es un texto fuente que incluye declaraciones de variables y sentencias, las cuales se ejecutarán de forma **secuencial**.

De forma **secuencial** quiere decir **ordenada en el tiempo**: *hasta que no acaba la ejecución de una sentencia, no comienza la siguiente*.

A la derecha, vemos un ejemplo de un programa secuencial en pseudo-código (una notación inspirada en los lenguajes Algol y Pascal):

```
Process EjemploSec ;
var x : integer := 0 ;
   y : integer := 2 ;
begin
   while x < 2 do begin
      x := x+1 ;
      y := 2*y+x+1 ;
   end
   print(x,y);
end
```

Sentencias, estados y procesos secuenciales.

Una **sentencia** es un trozo del texto de un programa el cual, al ejecutarse, modifica el valor de una o de varias variables del programa, para ello realiza uno o varios accesos de escritura a las mismas.

Un **estado** de un programa es el conjunto de valores de las variables del programa en un instante durante su ejecución.

Un **proceso secuencial** es una ejecución (en un tiempo finito) de un programa secuencial.

Durante dicha ejecución, se ejecutan las sentencias del programa, que cambian las variables desde un **estado inicial** hasta un **estado final**.

Accesos a variables en una sentencia

Un **acceso a una variable** es una operación realizada por el procesador por la cual lee o escribe dicha variable durante la ejecución de un programa.

- Consideramos únicamente accesos a variables de tipos *primitivos*: lógicos, enteros, carácter o flotantes (un acceso a una variable compuesta es una secuencia de accesos a variables primitivas).
- Cuando se ejecuta una sentencia, **se producen uno o varios de estos accesos**, los cuales se realizan también de forma **secuencial**, es decir: *hasta que no acaba uno, no puede comenzar el siguiente.*
- Las sentencias más simples posibles producen un único acceso de escritura, por ejemplo, `x:=34`.
- Las asignaciones con expresiones más complejas, y otros tipos de sentencias, producen típicamente más de un acceso.

Variables accedidas por una sentencia

El conjunto $V(S)$ de **variables accedidas por una sentencia** S es el conjunto de variables que se leen o escriben durante una ejecución de S. Ejemplos tomados del programa:

- La sentencia `x:=x+1` accede a la variable `x` (la lee y la escribe). El conjunto es:

$$V(\texttt{x:=x+1}) = \{\texttt{x}\}$$

- La sentencia `y:=2*y+x+1` accede a las variables `x` e `y` (lee ambas y escribe `y`). El conjunto es:

$$V(\texttt{y:=2*y+x+1}) = \{\texttt{x},\texttt{y}\}$$

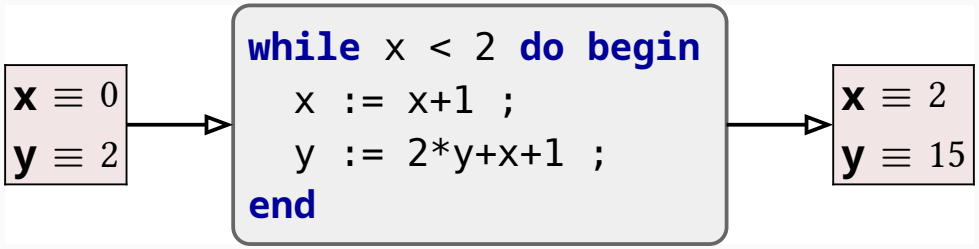
- La sentencia `print(x,y)` accede a las variables `x` e `y` (las lee).

$$V(\texttt{print(x,y)}) = \{\texttt{x},\texttt{y}\}$$

Traza y ejemplo (1/2)

La **historia** o **traza** de una ejecución de un proceso secuencial es la secuencia de estados por los que pasa el proceso durante dicha ejecución.

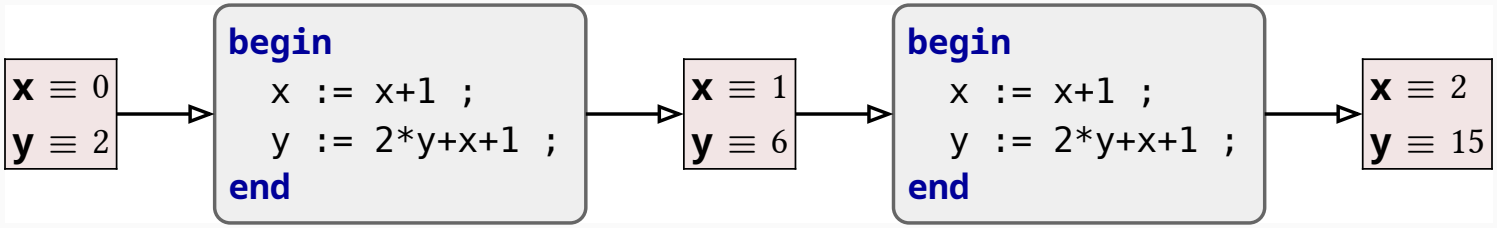
A modo de ejemplo, para analizar su traza, consideramos el programa secuencial que hemos visto. Se parte de un estado inicial (a la izquierda), se ejecuta la sentencia **`while`** (en el centro) y se llega a un estado final (a la derecha).



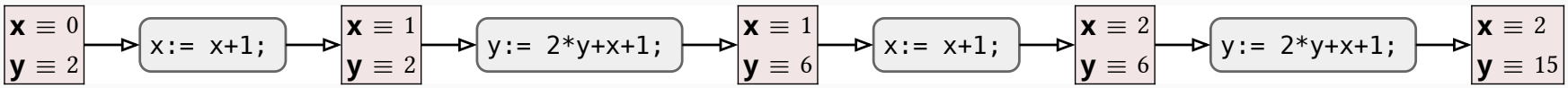
- Las sentencias tienen fondo gris y los estados fondo rojo.
- En este caso vemos una sentencia **`while`**, la cual a su vez está compuesta de otras sentencias.

Traza y ejemplo. (2/2)

La ejecución de la sentencia **`while`** en realidad se descompone en la doble ejecución de la sentencia bloque (entre **`begin`** y **`end`**), por tanto, se produce un estado intermedio (en el centro).



Cada ejecución del bloque se descompone a su vez en dos sentencias de asignación, así que la secuencia de estados detallada a nivel de asignaciones sería esta:



Tradución de expresiones aritméticas. Registros.

En el pseudo-código se pueden usar expresiones aritméticas enteras, flotantes o lógicas, como, por ejemplo, las expresiones enteras `x+1` o `2*y+x+1` que aparecen en el programa de ejemplo.

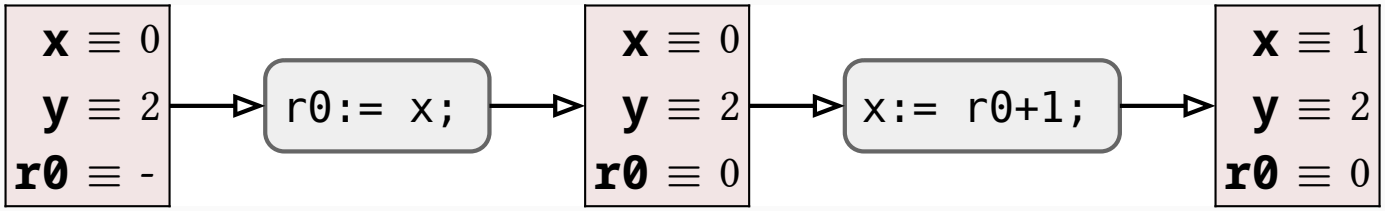
- Para poder evaluarlas, una CPU necesita que los operandos de los operadores aritmético-lógicos (`+`,`-`,`*`,**`and`**, **`or`**, etc...) se guarden en los llamados **registros** del procesador.
- A efectos de analizar el comportamiento, suponemos que cada proceso secuencial cuenta con una o varias variables propias (implícitamente declaradas), llamadas también **registros**, que se usan para evaluar las expresiones. Las llamamos `r0`, `r1`, `r2`, etc....
- Por cada referencia a una variable en una expresión, se debe hacer una lectura de la variable y una escritura en un registro.
- Los valores de los registros forman parte del estado del proceso secuencial.

Ejemplos de sentencias y sus accesos a memoria.

A modo de ejemplo, la sentencia de asignación `x:=x+1` se traduce al compilar en la sentencia compuesta `r0:=x ; x:=r0+1`, es decir, se hace:

1. `r0:=x` lectura de `x` y escritura en el registro `r0`
2. `x:=r0+1` lectura de `r0`, cálculo de `r0+1` y escritura sobre `x`.

A modo de ejemplo, partiendo del estado `(x==0,y==2)`, se produce esta traza:

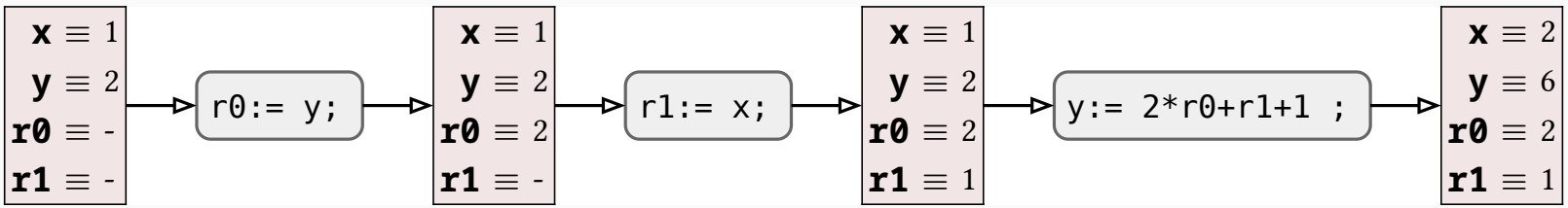


Traza de asignaciones usando dos registros

En otras expresiones más complejas, se necesitan más de un registro y más de una lectura, por ejemplo, para `y:=2*y+x+1` se necesitan dos registros (pues hay dos referencias a variables en la expresión a la derecha de `:=`), los pasos son:

1. `r0:=y` acceso de lectura a `y`, y escritura en el registro `r0`.
2. `r1:=x` acceso de lectura a `x`, y escritura en el registro `r1`.
3. `y:=2*r0+r1+1` evaluación de la expresión (lectura de `r0` y `r1`), seguida de acceso de escritura en `y`.

Partiendo del estado `(x==1,y==2)` se obtiene esta traza:



Sentencias compuestas y traza

La traza puede escribirse en forma de tabla, en la primera fila aparece el estado inicial, después, en cada fila aparece una sentencia y el estado después de ejecutarse.

A nivel de asignaciones, sería así:

Sentencia ejecutada	Estado	
	x	y
	0	2
`x := x+1`	1	2
`y := 2*y+x+1`	1	6
`x := x+1`	2	6
`y := 2*y+x+1`	2	15

Traza detallada con un acceso por fila (2/2)

Vemos la traza detallada, incluyendo las operaciones con registros:

Sentencia ejecutada	Estado			
	r0	r1	x	y
	-	-	0	2
`r0 := x`	0	-	0	2
`x := r0+1`	0	-	1	2
`r0 := y`	2	-	1	2
`r1 := x`	2	1	1	2
`y := 2*r0+r1+1`	2	1	1	6
`r0 := x`	1	-	1	6
`x := r0+1`	1	-	2	6
`r0 := y`	6	-	2	6
`r1 := x`	6	2	2	6
`y := 2*r0+r1+1`	6	2	2	15

El proceso secuencial como secuencia de accesos

En resumen:

> La ejecución de un proceso secuencial se puede modelar como **una secuencia finita y ordenada de accesos a las variables** (de tipos simples) declaradas explícitamente en el correspondiente programa, a partir del estado inicial.

- Suponemos que la secuencia es finita porque el proceso nunca entra en en un bucle infinito (siempre se llega a un estado final).
- Este modelo de un proceso secuencial es el que usaremos para **analizar el comportamiento de programas concurrentes en memoria compartida** (en la siguiente subsección).

Subsección 2.3.
Modelo abstracto de ejecución concurrente.

Programa concurrente.

Un **programa concurrente** es un texto fuente que incluye declaraciones de variables (algunas de ellas **compartidas**) y sentencias, de forma que dichas sentencias determinan cómo dos o más procesos secuenciales se pueden ejecutar **concurrentemente**.

Aquí ***concurrentemente*** quiere decir que la ejecución de cada sentencia de cada proceso secuencial **puede** solaparse en el tiempo con la ejecución de sentencias del resto de procesos secuenciales.

- Las **variables compartidas** son variables accesibles (para leer o escribir) por cualquiera de los procesos secuenciales que se ejecuten.
- Además de las variables compartidas, cada proceso puede tener sus propias **variables locales** y sus propios **registros** (en ambos casos son no accesibles por otros procesos).

Ejemplo de programa concurrente.

A modo de ejemplo, vemos aquí un programa concurrente (usando una de las posibles formas de escribirlo)

```
{ Variables compartidas }
var x : integer := 0 ;
    y : integer := 2 ;

{ Procesos secuenciales }
process NomUno ;
   var a : integer := 1 ;
begin
   x := x+1 ;
end
process NomDos ;
   var b : integer := 2 ;
begin
   y := y+x+1 ;
end
```

- Hay dos procesos secuenciales: `NomUno` y `NomDos`.
- Ambos acceden a las variables compartidas (`x` e `y`).
- Antes de ejecutar los procesos, se inicializan las variables compartidas.
- A partir de ahí, ambos procesos se ejecutan *concurrentemente*.
- El programa acaba cuando han acabado los dos procesos.

Variables locales y registros

En un programa concurrente:

- Cada proceso secuencial puede incluir declaraciones de sus propias **variables locales**: son variables que únicamente pueden ser accedidas por el proceso secuencial donde están declaradas. En el ejemplo:
  - La variable `a` es local al proceso `NomUno`
  - La variable `b` es local al proceso `NomDos`
- Dos variables locales de dos procesos secuenciales distintos podrían tener el mismo nombre (siguen siendo dos variables distintas).
- Cada proceso secuencial usa sus propios registros para poder ejecutar las sentencias aritmético-lógicas, (esos registros son equivalentes a variables locales, en el sentido de que únicamente son accesibles por su proceso y no los otros).

Notación abreviada para ejecución secuencial o concurrente.

Si S_A y S_B son dos sentencias, podemos hablar de la ejecución secuencial o la ejecución concurrente de ambas. Por simplicidad a veces usaremos una notación abreviada:

- Usaremos $S_A ; S_B$ para denotar la sentencia compuesta cuya ejecución consiste en ejecutar S_A completa, y después, cuando haya acabado, ejecutar S_B completa (es decir **secuencialmente**).
- Usaremos $S_A \| S_B$ para denotar la sentencia compuesta cuya ejecución consiste en ejecutar S_A y S_B de forma simultánea (es decir: **concurrentemente**). Esta sentencia acaba cuando hayan acabado tanto S_A como S_B.
- Ambas construcciones pueden extenderse a más de dos sentencias, por ejemplo $S_A ; S_B ; S_C$ o bien $S_A \| S_B \| S_C$
- La sentencia $S_A \| S_B$ es equivalente a $S_B \| S_A$, pero $S_A ; S_B$ no es igual a $S_B ; S_A$.

Accesos a variables compartidas: consistencia secuencial

Es necesario suponer que siempre se cumple esta propiedad:

Propiedad de **Consistencia Secuencial Estricta**:

- La totalidad de los accesos a las variables realizados por todos los procesos (desde el inicio hasta el fin) **ocurren en un orden determinado**, sin solaparse: hasta que no ha acabado un acceso, no comienza el siguiente (incluso si son dos accesos a variables distintas). Ese orden **puede ser distinto en cada ejecución**.
- En ese orden de todos los accesos, los accesos realizados por un proceso secuencial P aparecen en el mismo orden en el que P los realiza.
- El valor leído en una variable es siempre el último valor escrito en ella, o el valor inicial si ningún proceso ha escrito aun en la variable.

Estados de un programa concurrente

Puesto que suponemos que se cumple la propiedad anterior, podemos hablar de los estados de un programa concurrente:

> Un **estado** de un programa concurrente es el conjunto de valores de las variables (compartidas y locales de cada proceso secuencial) en un instante durante su ejecución.

Al igual que en un proceso secuencial, hay:

- Un estado inicial y otro final.
- Un estado distinto entre cada dos accesos consecutivos (según el orden conjunto), ahora esos dos accesos pueden ser del mismo proceso o de dos procesos distintos.

Por tanto, cabe hablar también de la **traza** de un programa concurrente.

Ejecución de sentencias de forma concurrente

Durante la ejecución de una sentencia S_1 en un programa concurrente:

- La sentencia producirá una secuencia ordenada de accesos a las variables compartidas
- Es posible que otra sentencia S_2 de otro proceso también acceda a alguna de esas mismas variables compartidas (si $V(S_1)$ tiene alguna variable compartida en común con $V(S_2)$).
- Si eso ocurre, el resultado de la ejecución de las sentencias S_1 y S_2 puede depender no solo de los valores de las variables y de las sentencias, sino también del orden en el que se mezclan los accesos.

Esta es una característica importante de la programación concurrente que debe ser tenida muy en cuenta para el diseño de programas concurrentes.

Ejemplo de ejecución concurrente de sentencias

Supongamos que dos procesos secuenciales comparten una variable entera `x` que vale 0 en un estado, y después de ese estado ambos procesos ejecutan la sentencia `x:=x+1` a la vez:

- Cada proceso usa su propio registro `r0`
- Cada proceso ejecuta estas dos sentencias en secuencia:
  1. lectura de la variable: `r0:=x`,
  2. cálculo del nuevo valor y escritura: `x:=r0+1`.
- Cuando hayan acabado ambas sentencias, el valor final de `x` dependerá del ordenamiento de los accesos.
- Ese orden puede ser cualquiera y por tanto hay **indeterminación**: no hay un único estado posterior, sino que puede haber varios (dependiendo del orden)

Sentencias atómicas y no atómicas (1/2)

Para simplificar el análisis y diseño de programas concurrentes, se introduce la noción de sentencia atómica:

Sentencia atómica

Una sentencia S de un proceso en un programa concurrente es **atómica** si y solo si siempre que se ejecuta (por un proceso secuencial P) no produce **estados intermedios** en $V(S)$ que sean **accesibles** para otros procesos secuenciales distintos de P.

- Consideramos **estado intermedio** a un estado que no es ni el inicial ni el final, y que está entre dos accesos a variables compartidas producidos por S.
- Un estado es **accesible para otros procesos** si contiene valores de variables compartidas que puedan ser escritos o leídos por otros procesos distintos de P.

Sentencias atómicas y no atómicas (2/2)

Durante la ejecución de una sentencia atómica S por parte de un proceso P:

- Los procesos distintos de P no pueden leer valores intermedios en las variables de $V(S)$, ni pueden modificar, mediante escrituras, esos valores intermedios.
- Es decir, los procesos distintos de P solo pueden acceder al estado previo o al estado posterior: **los posibles cambios de valores de las variables en $V(S)$ ocurren aparentemente *de una vez* para el resto de procesos.**
- Según la definición, **es atómica cualquier sentencia que únicamente hace un acceso a una variable compartida** (ya que nunca produce estados intermedios).
- Existen sentencias atómicas con estados intermedios que no son accesibles por otros procesos (lo vemos más adelante).

Ejemplos de sentencias atómicas (1/3).

Hay diversos tipos de sentencias atómicas, a modo de ejemplo (sin ser exhaustivos), vemos estos:

- Sentencias que no producen estados intermedios y únicamente realizan un acceso a una variable compartida, por ejemplo:
  - Escribir un valor en una variable compartida: `x:=34`.
  - Leer el valor de una variable compartida y escribirlo en un registro: `r0:=x`.
  - Leer el valor de un registro y escribirlo en una variable compartida: `x:=r0`.

  Estas sentencias típicamente se traducen en una única instrucción de código máquina, aunque ese detalle **es irrelevante para la concurrencia**: nos basta saber que son atómicas siempre y nos abstraemos de los detalles del hardware y el repertorio de instrucciones máquina.

Ejemplos de sentencias atómicas (2/3).

También son atómicas este otro tipo de sentencias (en adelante, suponemos que `x` e `y` son variables compartidas y `a` y `b` son variables locales):

- Sentencias con múltiples accesos a variables, pero **como mucho uno de ellos a una variable compartida**. Ejemplos:
  - Realizar varias operaciones con registros y/o variables locales, y además escribir una vez en una variable compartida, por ejemplo: `x:=2*r0+r1` o bien `x:=a+b`.
  - Igual, pero con una única lectura en una variable compartida, por ejemplo, `a:=2*b+x+1`, o bien `r0:=r1+x`.
  - Operaciones con uno o varios accesos, pero ninguno de ellos a variables compartidas, por ejemplo: `r0:=r0+1`, o bien `a:=2*b+a+1`.

Ejemplos de sentencias atómicas (3/3).

Una sentencia compuesta puede ser atómica, si cumple la definición, aunque realize muchas operaciones. Por ejemplo, estas dos sentencias bloque son atómicas (siendo a y b locales y x e y compartidas):

```
begin
   a := 2*a*a + 3*b + 4 ;
   b := b+1 ;
   x := 2*(a+b) ; { escribe x }
   b := b-1 ;
end
```

```
begin
   a := 2*a*a + 3*b + 4 ;
   b := b+1 ;
   a := 2*x*(a+b); { lee x }
   b := b-1 ;
end
```

- La sentencia de la izquierda accede a sus variables locales y produce una única escritura en una variable compartida. El resto de procesos únicamente podrán detectar un único cambio *atómico* del valor de la variable x.
- La sentencia de la derecha realiza una única lectura de una variable compartida (y cambia el estado de sus variables locales).

Ejemplos de sentencias no atómicas

Muchas sentencias son **no atómicas** (producen estados intermedios que sí pueden ser accedidos por otras hebras), esencialmente debido a que esas sentencias **hacen más de una operación de lectura y/o escritura en variables compartidas**. Por ejemplo:

- La sentencia `x:=x+1`, produce un estado intermedio, posterior a la lectura de `x` pero previo a la escritura en `x`.
- La sentencia `x:=x+y`, produce dos estados intermedios (cada uno inmediatamente posterior a una de las dos lecturas).
- Si una sentencia tiene dos o más lecturas de variables compartidas, es no atómica, aunque no tenga escrituras. Por ejemplo, la sentencia `a:=x+y`, produce un estado intermedio accesible en las variables compartidas, que ocurre entre la lectura de `x` y la de `y`.

Ejemplo de solapamiento de instrucciones atómicas (1/2)

Supongamos que dos procesos secuenciales P_A y P_B ejecutan concurrentemente las sentencias atómicas S_A (`x:=a+1`) y S_B (`b:=x+1`) partiendo de un estado en el que `x` (compartida), `a` (local de P_A) y `b` (local de P_B) valen todas 0:

- S_A escribe una vez en `x` y S_B lee una vez de `x`.
- El estado final solo depende del orden entre los dos accesos a `x` (no importa el orden de los accesos a registros y las variables locales).
- Por tanto, únicamente hay dos posibles estados finales.
  - Si se lee `x` y luego se escribe, al final `b==1`.
  - Si se escribe `x` y luego se lee, al final `b==2`.

En la siguiente diapositiva vemos dos trazas que dan lugar a los dos valores de `b` posibles. Escribe, el resto de trazas y verifica que el valor final solo depende del orden de los accesos a `x`.

Ejemplo de solapamiento de instrucciones atómicas (2/2)

Vemos dos trazas que dan lugar a los dos valores de b posibles:

P_A	P_B	r_A	a	r_B	b	x
		-	0	-	0	0
r_A:= a		0	0	-	0	0
x := r_A+1		0	0	-	0	1
	r_B:= x	0	0	1	0	1
	b := r_B+1	0	0	1	**2**	1

P_A	P_B	r_A	a	r_B	b	x
		-	0	-	0	0
r_A:= a		0	0	-	0	0
	r_B:= x	0	0	0	0	0
x := r_A+1		0	0	0	0	1
	b := r_B+1	0	0	0	**1**	1

Solapamiento de sentencias atómicas

En general, si suponemos que dos procesos secuenciales distintos ejecutan concurrentemente cada uno de ellos una sentencia atómica cualquiera (S_0 uno y S_1 el otro), a partir de una estado E, entonces:

- Puesto que S_0 y S_1 **son atómicas**, **la ejecución concurrente de ambas es *equivalente* a la secuencial**, es decir el estado posterior **siempre será uno cualquiera** de estos dos:
  - Estado resultado de ejecutar S_0 completa, seguida de S_1.
  - Estado resultado de ejecutar S_1 completa, seguida de S_0.

Es decir: si S_0 y S_1 son ambas atómicas, entonces cada vez que se ejecuta la sentencia $S_0 \| S_1$ el resultado será equivalente a ejecutar $S_0 \,;\, S_1$ o bien a ejecutar $S_1 \,;\, S_0$.

Por tanto, en general, **podemos analizar el comportamiento de un programa concurrente suponiendo que las instrucciones atómicas no se solapan** (aunque realmente sí lo hagan).

Interfoliación de sentencias atómicas

Supongamos un programa concurrente C con dos procesos secuenciales,

- uno ejecuta la secuencia de instr. atómicas $A_1\ A_2\ A_3\ A_4\ A_5$,
- y otro ejecuta la secuencia de instr. atómicas $B_1\ B_2\ B_3\ B_4\ B_5$.

Entonces, al ejecutar el programa concurrente C, **puede ocurrir cualquier *interfoliación* de sentencias atómicas** que respete el orden dentro de cada proceso secuencial.

Aquí vemos algunos ejemplos de interfoliaciones posibles:

$A_1\ A_2\ A_3\ A_4\ A_5\ B_1\ B_2\ B_3\ B_4\ B_5$
$B_1\ B_2\ B_3\ B_4\ B_5\ A_1\ A_2\ A_3\ A_4\ A_5$
$A_1\ B_1\ A_2\ B_2\ A_3\ B_3\ A_4\ B_4\ A_5\ B_5$
$B_1\ B_2\ A_1\ B_3\ B_4\ A_2\ B_5\ A_3\ A_4\ A_5$
$\ldots$

El número de posibles interfoliaciones

El proceso P_1 ejecuta n_1 instr. atómicas, y el proc. P_2 ejecuta n_2:

- Hay tantas posibles interfoliaciones de P_1 y P_2 como posibles subconjuntos (de n_1 elementos) del conjunto $\{1, 2, \ldots, n_1 + n_2\}$ (es igual si consideramos subconjuntos de n_2 elementos). Luego el núm. de interfoliaciones es este *coeficiente binomial*:

$$\left(\begin{array}{c} n_1 + n_2 \\ n_1 \end{array} \right) = \frac{(n_1 + n_2)!}{n_1!\, n_2!}$$

- Si tenemos k procesos se usa el *coeficiente multinomial*:

$$\left(\begin{array}{c} n_1 + n_2 + \cdots + n_k \\ n_1,\, n_2,\, \ldots,\, n_k \end{array} \right) = \frac{(n_1 + n_2 + \cdots + n_k)!}{n_1!\, n_2! \cdots n_k!}$$

- Por ejemplo, para $n_1 = 10$ y $n_2 = 15$ hay $3.268.760$ interf.

Conclusión: **el número de posibles interfoliaciones es muy elevado incluso para programas cortos**

Abstracción

El modelo basado en el estudio de todas las posibles secuencias de ejecución entrelazadas de los procesos constituye una **abstracción**:

- Se consideran exclusivamente las **características relevantes** que determinan el resultado del cálculo
- Esto permite **simplificar** el análisis y diseño de los programas concurrentes.

Se **ignoran los detalles no relevantes** para el resultado, por ejemplo:

- Las áreas de memoria asignadas a los procesos.
- Los registros particulares que están usando.
- El costo de los cambios de contexto entre procesos.
- La política del S.O. relativa a asignación de CPU.
- Las diferencias entre entornos multiprocesador o monoprocesador.

Velocidad de ejecución. Hipótesis del progreso finito.

Progreso Finito

No se puede hacer ninguna suposición acerca de las velocidades absolutas/relativas de ejecución de los procesos, salvo que es mayor que cero.

Un programa concurrente se entiende en base a sus componentes (procesos) y sus interacciones, sin tener en cuenta el entorno de ejecución.

Ejemplo: Un disco es normalmente más lento que una CPU, pero no podemos suponer que siempre es así para diseñar un programa.

Si se hicieran suposiciones temporales:

- Sería difícil detectar y corregir fallos
- La corrección dependería de la configuración de ejecución, que puede cambiar

Hipótesis del progreso finito

Si se cumple la hipótesis, la velocidad de ejecución de cada proceso será no nula, lo cual tiene estas dos consecuencias:

Punto de vista global

Durante la ejecución de un programa concurrente, en cualquier momento existirá al menos 1 proceso preparado, es decir, eventualmente se permitirá la ejecución de algún proceso.

Punto de vista local

Cuando un proceso concreto de un programa concurrente comienza la ejecución de una sentencia, completará la ejecución de la sentencia en un intervalo de tiempo finito.

Notación para expresar ejecución concurrente. Tipos.

Usaremos una notación (la llamamos **pseudo-código**) para expresar el código y la sincronización de los distintos procesos secuenciales que forman un programa concurrente.

Distinguimos dos tipos de sistemas concurrentes en función de las posibilidades para especificar cuáles son sus procesos:

Sistemas Estáticos

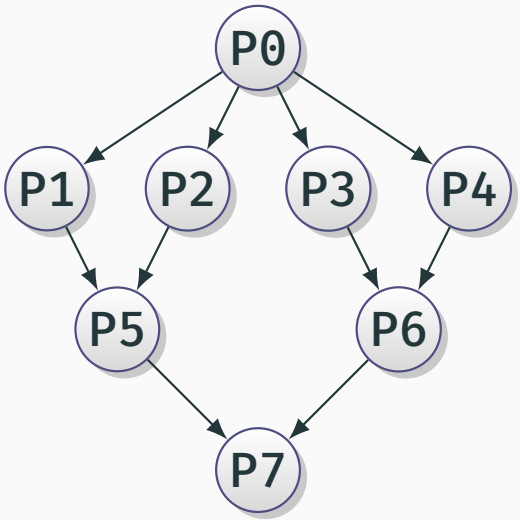
- Número de procesos fijado en el fuente del programa.
- Los procesos se activan al lanzar el programa.
- Ejemplo: *Message Passing Interface* (*MPI-1*).

Sistemas Dinámicos

- Número variable de procesos/hebras que se pueden activar en cualquier momento de la ejecución.
- Ejemplos: *OpenMP*, *PThreads*, *Java Threads*, *MPI-2*.

Grafo de Sincronización

Un **Grafo de Sincronización** es un Grafo Dirigido Acíclico (DAG) donde cada nodo representa una **secuencia de sentencias del programa** (una **actividad**).



Dadas dos actividades A y B, una arista (flecha) desde A hacia B significa que B **no puede comenzar su ejecución antes de que A termine**.

El grafo muestra las restricciones de precedencia que determinan cuándo una actividad puede empezar en un programa.

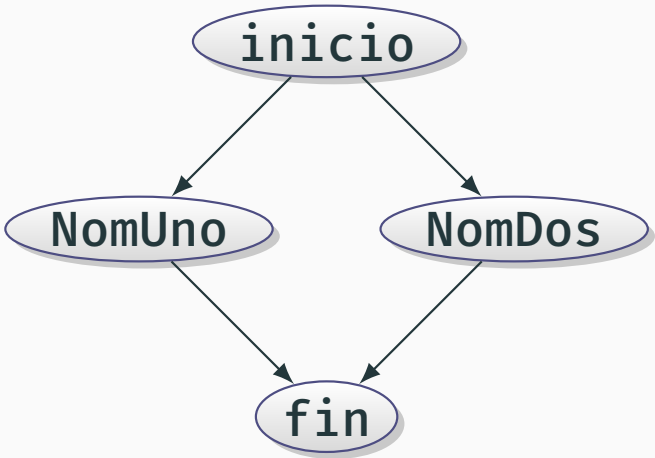
Ejemplo: B lee una variable compartida que ha debido ser escrita antes por A.

Definición estática de procesos

El número de procesos (arbitrario) y el código que ejecutan no cambian entre ejecuciones. Cada proceso se asocia con su identificador y su código mediante la palabra clave **`process`**.

```
var ...  { vars. compartidas }

process NomUno ;
var  ... { vars. locales }
begin
   ....  { código }
end
process NomDos ;
var .... { vars. locales }
begin
   ....  { codigo }
end
...      { otros procesos }
```



El programa acaba cuando acaban todos los procesos. Las vars. compartidas se inicializan antes de que comienzen los procesos.

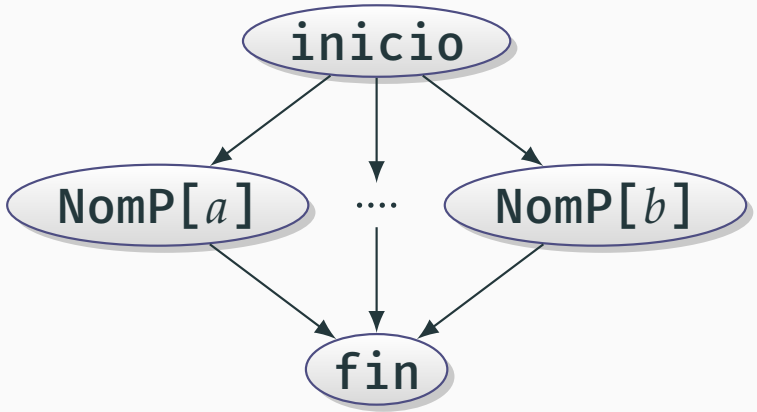
Definición estática de vectores de procesos

Se pueden usar definiciones estáticas de grupos de procesos similares que sólo se diferencian en el valor de una constante (**vectores de procesos**)

```
var ... { vars. compartidas }

process NomP[ ind : a..b ] ;
var  ... { vars. locales }
begin
 ..... { código }
 ..... { (ind vale a,a+1,...,b) }
end

...   { otros procesos }
```



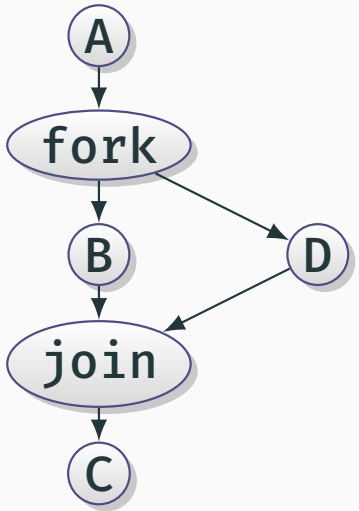
- En cada caso, a y b se traducen por dos constantes concretas (el valor de a será típicamente 0 ó 1).
- El número total de procesos será $b-a+1$ (se supone que $a <= b$).

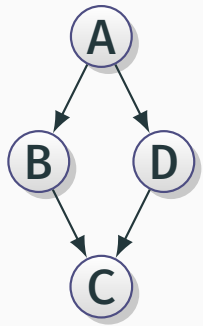
Creación de procesos no estructurada con fork-join.

- **fork**: sentencia que especifica que la rutina nombrada puede comenzar su ejecución al mismo tiempo que comienza la sentencia siguiente (*bifurcación*).
- **join**: sentencia que espera la terminación de la rutina nombrada antes de comenzar la sentencia siguiente (*unión*).

```
procedure P1 ;
begin
   A ;
   fork P2 ;
   B ;
   join P2 ;
   C ;
end
```

```
procedure P2 ;
begin
   D ;
end
```





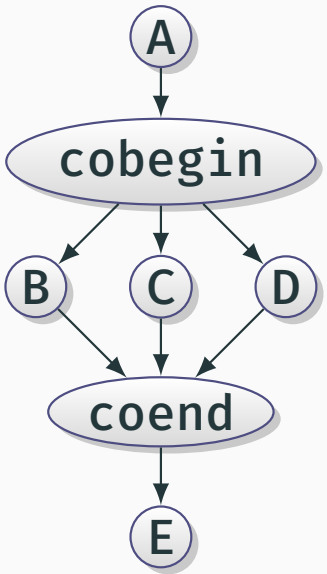
- **Ventajas:** práctica y potente, creación dinámica.
- **Inconvenientes:** no estructuración, difícil comprensión de los programas.

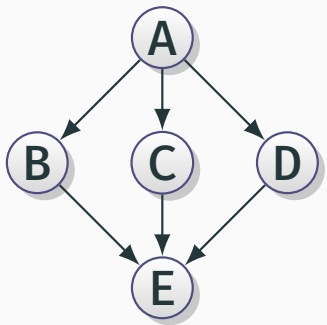
Creación de procesos estructurada con cobegin-coend (1/2)

Los bloques delimitados por **cobegin-coend** implican paralelismo potencial de varias sentencias (producen ejecución concurrente):

- Tras **cobegin** empezarán todas ellas a ejecutarse.
- En **coend** se espera a que todas terminen.

```
begin
   A ;
   cobegin
      B ; C ; D ;
   coend
   E ;
end
```





- **Ventajas**: impone estructura: 1 única entrada y 1 única salida $\Longrightarrow$ más fácil de entender.
- **Inconveniente**: menor potencia expresiva que **fork-join**.

Creación de procesos estructurada con cobegin-coend (2/2)

El uso de **cobegin-coend** es equivalente a las dos barras paralelas, es decir, una sentencia como esta:

```
cobegin
   A ; B ; C ;
coend
```

Es equivalente esta sentencia compuesta con las barras:

```
A || B || C
```

- En ambos casos, la ejecución de la sentencia compuesta supone la creación (dinámica) de tres procesos secuenciales, los tres dentro del programa concurrente que ejecuta dicha sentencia compuesta.
- Al acabar los tres procesos, acaba la sentencia.

Sección 3.
Exclusión mutua y sincronización.

3.1. Concepto de exclusión mutua
3.2. Condición de sincronización

Exclusión mutua y sincronización

Según el modelo abstracto, los diversos procesos secuenciales dentro de un programa concurrentes ejecutan sus instrucciones atómicas de forma que:

- En principio es completamente arbitraria e impredecible la interfoliación en el tiempo de sus respectivas secuencias de instrucciones atómicas.
- Sin embargo, los procesos secuenciales **no son independientes entre sí** (es decir, son **cooperativos**): necesariamente debe haber cierta coordinación entre ellos, ya que, si no se necesitase ninguna coordinación en absoluto, bastaría con ejecutar cada proceso secuencial de forma aislada.
- Esa coordinación necesaria implica que **algunas de las posibles formas de combinar las secuencias no sean válidas**.

Condición de sincronización. Exclusión Mutua

Si no todas las posibles interfoliaciones son permisibles, entonces

- Diremos que hay una **Condición de Sincronización** cuando tenemos que imponer alguna restricción (condición) sobre el orden en el que se pueden interfoliar las instrucciones atómicas de distintos procesos.
- Un caso particular es la **Exclusión Mutua** (EM), son secuencias finitas de instrucciones que deben ejecutarse de principio a fin por un único proceso, sin que a la vez otro proceso las esté ejecutando también.
- Además de la EM, existen otros tipos de condiciones de sincronización que veremos en esta asignatura.

Subsección 3.1.
Concepto de exclusión mutua.

Exclusión mutua

La restricción se refiere a una o varias secuencias de instrucciones consecutivas que aparecen en el texto de uno o varios procesos.

- Al conjunto de dichas secuencias de instrucciones se le denomina **sección crítica** (SC).
- Ocurre **exclusión mutua** (EM) cuando los procesos solo funcionan correctamente si, en cada instante de tiempo, **hay como mucho uno de ellos ejecutando cualquier instrucción de la sección crítica**.

Es decir, el solapamiento de las instrucciones debe ser tal que cada secuencia de instrucciones de la SC se ejecuta como mucho por un proceso de principio a fin, sin que (durante ese tiempo) otros procesos ejecuten ninguna de esas instrucciones ni otras de la misma SC.

Ejemplos de exclusión mutua

En aplicaciones concurrentes a menudo se hace necesario resolver el problema de la exclusión mutua, para actualizaciones concurrentes de datos en memoria compartida, por ejemplo:

- Añadir y/o eliminar nodos de forma concurrente por varios procesos a una lista enlazada de nodos o en general, accesos a cualquier estructura de datos dinámica, cuando cada acceso requiere múltiples instrucciones atómicas.
- Modificar o consultar un contador en memoria compartida, por ejemplo, en un sistema de reservas de asientos para eventos, cuando queremos que varios procesos puedan decrementar el contador de asientos libres.

Un ejemplo sencillo de exclusión mutua

Para ilustrar el problema de la EM, veremos un ejemplo sencillo que usa una variable entera (`x`) en memoria compartida y operaciones aritméticas elementales.

- La sección crítica está formada por todas las secuencias de instrucciones máquina que se obtienen al traducir (compilar) operaciones de escritura (o lectura y escritura) de la variable (p.ej., asignaciones como `x:=x+1` o `x:=4*z`).
- Veremos que si varios procesos ejecutan las instrucciones máquina de la sección crítica de forma simultánea, los resultados de este tipo de asignaciones son **indeterminados**.

Aquí, el término *indeterminado* indica que para cada valor inicial de `x`, existe un conjunto de posibles valores de `x` al finalizar. El valor concreto que toma `x` depende la interfolicación concreta que ocurre.

Traducción y ejecución de asignaciones

Supongamos dos procesos (P_A y P_B) que ejecutan cada uno `x:=x+1` (que forma la sección crítica). Sabemos que

- ambos procesos comparten `x` y cada uno realiza dos accesos a esa variable (primero la lee y luego la escribe),
- la interfoliación (de las sentencias atómicas ejecutadas) es arbitraria.

Una traducción típica a código máquina sería equivalente a estas **dos sentencias atómicas**:

1. `r := x`
   leer el valor de la variable compartida `x` y escribirlo en un registro `r` (propio del proceso).
2. `x := r+1`
   calcular el valor `r+1` y escribirlo en la variable compartida `x`.

Posibles secuencias de instrucciones

Suponemos que inicialmente `x` vale 0 y ambos procesos ejecutan la asignación. El proceso A usa el registro $\mathtt{r}_A$ y el proceso B usa $\mathtt{r}_B$. Vemos dos posibles trazas de sentencias atómicas distinguiendo qué proceso ejecuta cada sentencia:

P_A	P_B	x
$\mathtt{r}_A$:= x		0
x := $\mathtt{r}_A$+1		1
	$\mathtt{r}_B$:= x	1
	x := $\mathtt{r}_B$+1	2

P_A	P_B	x
$\mathtt{r}_A$:= x		0
	$\mathtt{r}_B$:= x	1
x := $\mathtt{r}_A$+1		1
	x := $\mathtt{r}_B$+1	1

- Partiendo de x$==0$, al final x puede valer 1 o 2, dependiendo de la interfoliación, la cual es impredecible.
- Es muy posible que en una aplicación el **valor final 1 sea considerado incorrecto**, ya que se han ejecutado dos incrementos a partir del valor 0.

Notación para garantizar atomicidad.

En nuestra notación de pseudo-código, podemos escribir sentencias indicando que se deben de ejecutar de forma atómica, usando los caracteres < y >.

- Sea S una sentencia cualquiera, entonces $< S >$ es otra sentencia *equivalente* a S pero que **es atómica por definición**.
- Aquí *equivalente* significa que hace los mismos accesos a memoria y los mismos cálculos que S.
- Los cambios de estado que pueda hacer $< S >$ en variables compartidas **ocurren aparentemente de una sola vez para los otros procesos secuenciales**.

La sentencia $< S >$ es de un nuevo tipo de sentencia atómica, ya que puede hacer más de un acceso a variables compartidas, **pero los posibles estados intermedios de esas variables compartidas no son accesibles a los otros procesos secuenciales**.

Ejemplo de sentencias atómicas

Aquí vemos, a modo de ejemplo, dos sentencias concurrentes parecidas, excepto que la de la derecha usa la notación que se acaba de introducir (la variable x es compartida):

```
{ instr. no atómicas }
begin
   x := 0 ;
   cobegin
      x:= x+1 ;
      x:= x-1 ;
   coend
end
```

```
{ instr. atómicas }
begin
   x := 0 ;
   cobegin
      < x:= x+1 > ;
      < x:= x-1 > ;
   coend
end
```

- En el código de la izquierda, al acabar, x puede tener un valor cualquiera del conjunto $\{-1, 0, 1\}$.
- A la derecha, x finaliza con seguridad con el valor 0.

Actividad de clase (1/2)

Otro ejemplo sería un programa concurrente en el cual varios procesos comparten dos variables enteras `x` e `y`, y queremos que siempre se cumpla `x+y == 10`.

Dado un valor n cualquiera (puede ser una constante o una variable local), para asignarle n a la variable `x` (y actualizar adecuadamente `y`) podríamos usar alguna de estas tres sentencias alternativas:

```
{ Opción 1 }
begin
x:= n;
y:= 10-n;
end
```

```
{ Opción 2 }
begin
<x:= n>;
<y:= 10-n>;
end
```

```
{ Opción 3 }
begin
< x:= n;
   y:= 10-n; >
end
```

Para cada opción: describe razonadamente si, al ejecutarse la sentencia bloque en un estado previo con `x+y==10`, se garantiza que siempre en el estado posterior se cumplirá `x+y==10`.

Actividad de clase (2/2)

Para verificar que la suma de ambas variables siempre es 10, escribimos código que imprime dicha suma.

Usamos una variable local `a` y alguna de las siguientes sentencias alternativas:

```
{ Opción 1 }
begin
a:= x+y;
print(a);
end
```

```
{ Opción 2 }
begin
< a:= x+y >;
print(a);
end
```

```
{ Opción 3 }
begin
< a:= x+y;
   print(a); >
end
```

Para cada opción: describe razonadamente si, al ejecutarse la sentencia bloque en un estado previo con `x+y==10`, se garantiza que siempre se imprimirá el valor 10.

Además: ¿ es atómica la sentencia **`begin-end`** de la opción 2 ?

Subsección 3.2.
Condición de sincronización.

Condición de sincronización.

En general, en un programa concurrente compuesto de varios procesos, una **condición de sincronización** establece que:

> *no son correctas todas las posibles interfoliaciones de las secuencias de instrucciones atómicas de los procesos.*

- esto ocurre típicamente cuando, en un punto concreto de su ejecución, uno o varios procesos deben esperar a que se cumpla una determinada condición global (depende de varios procesos).

Veremos un ejemplo sencillo de condición de sincronización en el caso en que los procesos puedan usar variables comunes para comunicarse (memoria compartida). En este caso, los accesos a las variables no pueden ordenarse arbitrariamente (p.ej.: leer de ella antes de que sea escrita)

Ejemplo de sincronización. Productor-Consumidor

Un ejemplo típico es el de dos procesos cooperantes en los cuales uno de ellos (productor) produce una secuencia de valores (p.ej. enteros) y el otro (consumidor) usa cada uno de esos valores. La comunicación se hace vía la variable compartida **x**:

```
{ variables compartidas }
var x : integer ; { contiene cada valor producido }
```

```
{ Proceso productor: calcula 'x' }
process Productor ;
   var a : integer ; { no compartida }
begin
   while true do begin
      { calcular un valor }
      a := ProducirValor() ;
      { escribir en mem. compartida }
      x := a ; { sentencia E }
   end
end
```

```
{ Proceso Consumidor: lee 'x' }
process Consumidor ;
   var b : integer ; { no compartida }
begin
   while true do begin
      { leer de mem. compartida }
      b := x ; { sentencia L }
      { utilizar el valor leido }
      UsarValor(b) ;
   end
end
```

Secuencias correctas e incorrectas

Los procesos descritos solo funcionan como se espera si el orden en el que se entremezclan las sentencias elementales etiquetadas como E (escritura) y L (lectura) es: $E, L, E, L, E, L, \ldots$.

- $L, E, L, E, \ldots$ es incorrecta: se hace una lectura de x previa a cualquier escritura (se lee valor indeterminado).
- $E, L, E, E, L, \ldots$ es incorrecta: hay dos escrituras sin ninguna lectura entre ellas (se produce un valor que no se lee).
- $E, L, L, E, L, \ldots$ es incorrecta: hay dos lecturas de un mismo valor, que por tanto es usado dos veces.

La secuencia válida asegura la condición de sincronización:

- Consumidor no lee hasta que Productor escriba un nuevo valor en x (cada valor producido es usado una sola vez).
- Productor no escribe un nuevo valor hasta que Consumidor lea el último valor almacenado en x (ningún valor producido se pierde).

Sección 4.
Propiedades de los sistemas concurrentes.

Concepto de corrección de un programa concurrente

Propiedad de un programa concurrente: Atributo del programa que es cierto para todas las posibles secuencias de interfoliación (historias del programa).

Hay 2 tipos:

- **Propiedad de seguridad** (*safety*).
- **Propiedad de vivacidad** (*liveness*).

Propiedades de Seguridad (*Safety*)

Son condiciones que **deben cumplirse en cada instante**, del tipo: *nunca pasará nada malo*.

- Requeridas en especificaciones estáticas del programa.
- Son fáciles de demostrar y para cumplirlas se suelen restringir las posibles interfoliaciones.

Ejemplos:

- **Exclusión mutua:** 2 procesos nunca entrelazan ciertas subsecuencias de operaciones.
- **Ausencia Interbloqueo (*Deadlock-freedom*):** Nunca ocurrirá que los procesos se encuentren esperando algo que nunca sucederá.
- **Propiedad de seguridad en el Productor-Consumidor:** El consumidor debe consumir todos los datos producidos por el productor en el orden en que se van produciendo.

Propiedades de Vivacidad (*Liveness*)

Son propiedades que **deben cumplirse eventualmente**, del tipo: *realmente sucede algo bueno*.

- Son propiedades dinámicas, más difíciles de probar.

Ejemplos:

- **Ausencia de inanición (*starvation-freedom*)**: Un proceso o grupo de procesos no puede ser indefinidamente pospuesto. En algún momento, podrá avanzar.
- **Equidad (*fairness*)**: Tipo particular de prop. de vivacidad. Un proceso que desee progresar debe hacerlo con justicia relativa con respecto a los demás. Más ligado a la implementación y a veces incumplida: existen distintos grados.

Sección 5.
Verificación de programas concurrentes.

5.1. Introducción
5.2. Enfoque axiomático.

Subsección 5.1. Introducción.

Introducción. Pruebas simples. Limitaciones.

¿ Cómo demostrar que un programa cumple una determinada propiedad ?

- **Posibilidad:** realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad.
- **Problema**: Sólo permite considerar un número limitado de historias (interfoliaciones) de ejecución y no demuestra que no existan casos indeseables.
- **Ejemplo**: Comprobar que el proceso P produce al final $x = 3$:

```
process P ;
   var x : integer := 0 ;
cobegin
   x := x+1 ; x := x+2 ;
coend
```

(hay varias historias que llevan a `x==1` o `x==2`, pero estas historias podrían no ocurrir en unas pocas ejecuciones).

Enfoque operacional (análisis exhaustivo)

- **Enfoque operacional**: Análisis exhaustivo de casos. Se chequea la corrección de todas las posibles historias.
- **Problema**: Su utilidad está muy limitada cuando se aplica a programas concurrentes complejos ya que el número de interfoliaciones crece **exponencialmente** con el número de instrucciones de los procesos.
- Para el sencillo programa P (2 procesos, 3 sentencias atómicas por proceso) habría que estudiar 20 historias diferentes.

Subsección 5.2.
Enfoque axiomático..

Verificación. Enfoque axiomático.

- Se define un *sistema lógico formal* que permite establecer propiedades de programas en base a axiomas y reglas de inferencia.
- Se usan fórmulas lógicas (asertos) para caracterizar un conjunto de estados.
- Las sentencias atómicas actúan como *transformadores de predicados* (asertos). Los teoremas en la lógica tienen la forma:

$$\{P\} \quad \mathbf{S} \quad \{Q\}$$

"Si la ejecución de la sentencia S empieza en algún estado en el que es verdadero el predicado P (*precondición*), entonces el predicado Q (*poscondición*) será verdadero en el estado resultante."

- **Menor Complejidad**: El trabajo que conlleva la prueba de corrección es proporcional al número de sentencias atómicas en el programa.

Invariante global

- **Invariante global**: Predicado que referencia variables globales siendo cierto en el estado inicial de cada proceso y manteniéndose cierto ante cualquier asignación dentro de los procesos.
- En una solución correcta del Productor-Consumidor, un invariante global sería:

$$consumidos \leq producidos \leq consumidos + 1$$

Bibliografía del tema 1.

Para más información, ejercicios, bibliografía adicional, se puede consultar:

1.1. **Conceptos básicos y Motivación**
Palma (2003), capítulo 1.

1.2. **Modelo abstracto y Consideraciones sobre el hardware**
Ben-Ari (2006), capítulo 2. Andrews (2000) capítulo 1. Palma (2003) capítulo 1.

1.3. **Exclusión mutua y sincronización**
Palma (2003), capítulo 1.

1.4. **Propiedades de los Sistemas Concurrentes**
Palma (2003), capítulo 1.

1.5. **Verificación de Programas concurrentes**
Andrews (2000), capítulo 2.

Fin de la presentación.