UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos: Tema 1. Introducción.

Carlos Ureña / Jose M. Mantas / Pedro Villar

2017-18

Grado en Ingeniería Informática / Grado en Ingeniería Informática y Matemáticas.
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 trabajen juntos.
- **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:

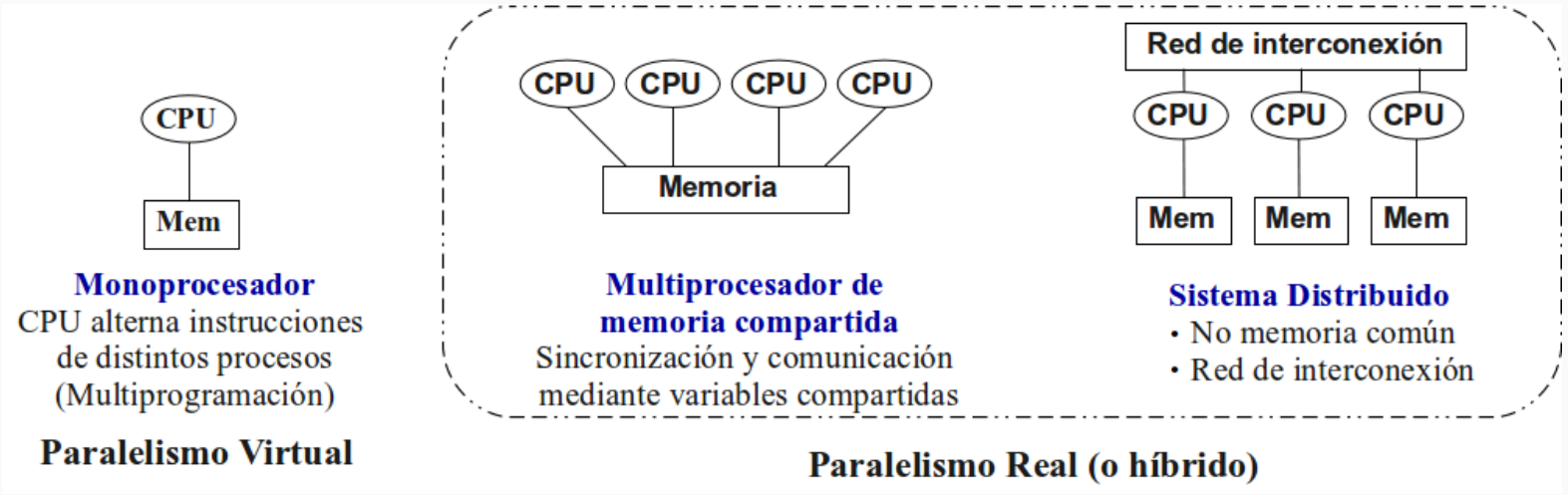
- **Servidor web para reserva de vuelos:** Es más natural, considerar cada petición de usuario como un proceso e implementar políticas para evitar situaciones conflictivas (permitir superar el límite de reservas en un vuelo).
- **Simulador del comportamiento de una gasolinera:** Es más sencillo considerar 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.

Sección 2.
Modelo abstracto y consideraciones sobre el hardware.

2.1. Consideraciones sobre el hardware
2.2. Modelo Abstracto de concurrencia

Subsección 2.1.
Consideraciones sobre el hardware.

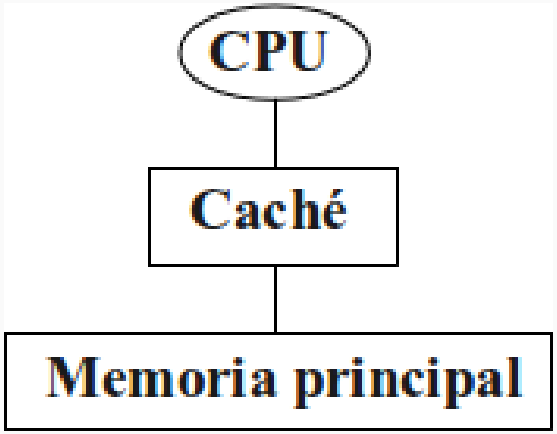
Modelos de arquitecturas para programación concurrente



Mecanismos de implementación de la concurrencia

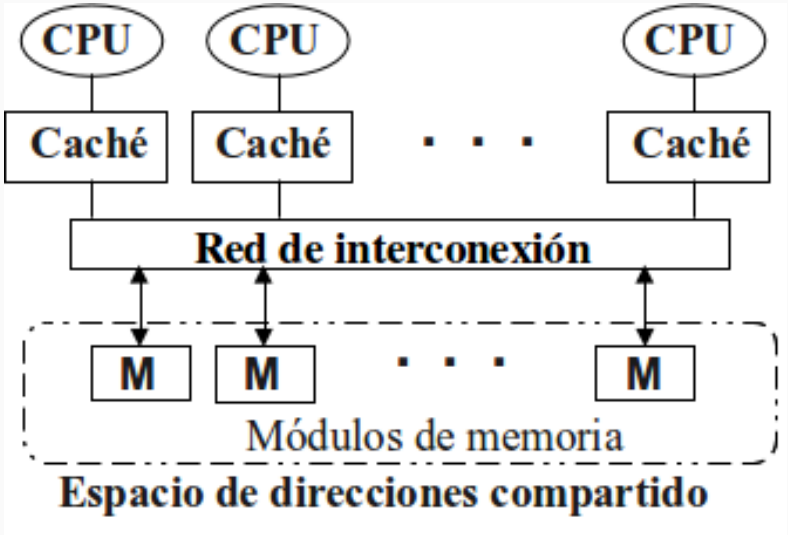
- Dependen fuertemente de la arquitectura.
- Consideran una *máquina virtual* que representa un sistema (multiprocesador o sistema distribuido), proporcionando base homogénea para ejecución de los procesos concurrentes.
- El tipo de paralelismo afecta a la eficiencia pero no a la corrección.

Concurrencia en sistemas monoprocesador



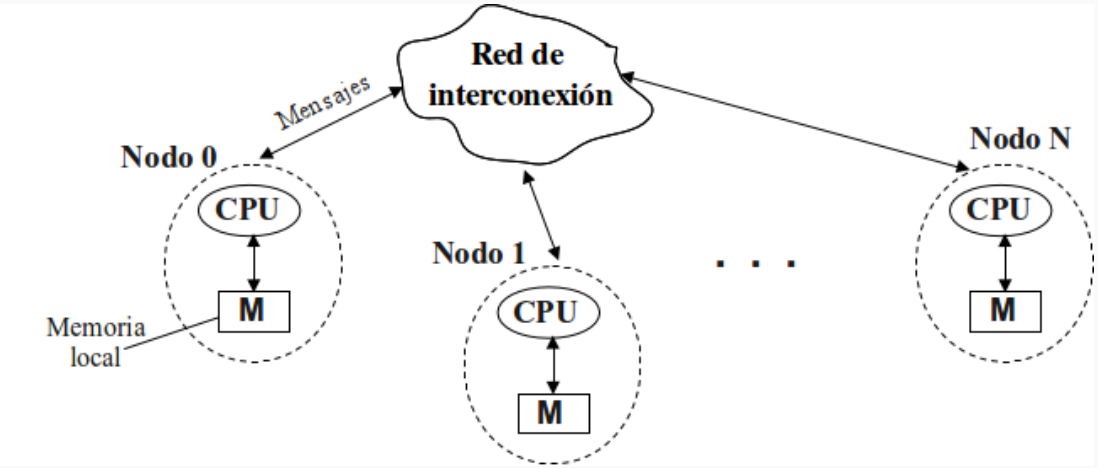
- **Multiprogramación:** El sistema operativo gestiona cómo múltiples procesos se reparten los ciclos de CPU.
- Mejor aprovechamiento CPU.
- Servicio interactivo a varios usuarios.
- Permite usar soluciones de diseño concurrentes.
- Sincronización y comunicación mediante variables compartidas.

Concurrencia en multiprocesadores de memoria compartida



- Los procesadores pueden compartir o no físicamente la misma memoria, pero 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 muticore.

Concurrencia en sistemas distribuidos



- No existe un 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 concurrencia.

Sentencias atómicas y no atómicas

Sentencia atómica (indivisible)

Una sentencia o instrucción de un proceso en un programa concurrente es **atómica** si siempre se ejecuta de principio a fin sin verse *afectada* (durante su ejecución) por otras sentencias en ejecución de otros procesos del programa.

- No se verá afectada cuando el *funcionamiento* de dicha instrucción **no dependa nunca** de como se estén ejecutando otras instrucciones.
- El funcionamiento de una instrucción se define por su efecto en el *estado de ejecución* del programa justo cuando acaba.
- El estado de ejecución esta formado por los valores de las variables y los registros de todos los procesos.

Ejemplos de sentencias atómicas.

A modo de ejemplo de instrucciones atómicas, cabe citar muchas de las instrucciones máquina del repertorio de un procesador, por ejemplo estas tres:

- Leer una celda de memoria (una variable de un tipo simple `x`) y cargar su valor en ese momento en un registro (`r`) del procesador.
- Incrementar el valor de un registro `r` (u otras operaciones aritméticas entre registros).
- Escribir el valor de un registro `r` en una celda de memoria.

Sentencias atómicas: ejemplo

El resultado de estas instrucciones **no depende nunca** de otras intrucciones que se estén ejecutando concurrentemente. Al finalizar, la celda de memoria o el registro (donde se escribe) tomará un valor concreto predecible siempre a partir del estado al inicio (el estado justo al acabar está **determinado**).

- Si en un estado el registro $\mathbf{r}$ tiene el valor v (un entero, p.ej.), y en ese estado se ejecuta la instrucción de escritura de $\mathbf{r}$ en una variable x, entonces en el estado final sabemos que x vale v.
- Lo anterior se puede asegurar incluso aunque haya otros procesos accediendo a la variable x
- Por tanto, el estado al acabar está bien definido como una función del estado al inicio.

Ejemplos de sentencias no atómicas.

La mayoría de las sentencias en lenguajes de alto nivel son típicamente no atómicas, por ejemplo, la sentencia:

```
x := x+1 ; { incrementa el valor de la variable entera en una unidad }
```

Para ejecutarla , el compilador o intérprete usa una secuencia de tres sentencias como esta:

1. leer el valor de `x` y cargarlo en un registro `r` del procesador
2. incrementar en un unidad el valor almacenado en el registro `r`
3. escribir el valor del registro `r` en la variable `x`

El resultado (es decir, el valor que toma `x` justo al acabar) **depende** de que haya o no haya otras sentencias ejecutándose a la vez y escribiendo simultáneamente sobre la variable `x`. Hay **indeterminación** (no se puede predecir el estado final a partir del inicial).

Interfoliación de sentencias atómicas

Supongamos que definimos un programa concurrente C compuesto de dos procesos secuenciales P_A y P_B que se ejecutan a la vez.

La ejecución de C puede dar lugar a cualquiera de las posibles mezclas (**interfoliaciones**) de sentencias atómicas de P_A y P_B.

Pr.	Posibles secuencias de instr. atómicas
P_A	$A_1\ A_2\ A_3\ A_4\ A_5$
P_B	$B_1\ B_2\ B_3\ B_4\ B_5$
C	$A_1\ A_2\ A_3\ A_4\ A_5\ B_1\ B_2\ B_3\ B_4\ B_5$
C	$B_1\ B_2\ B_3\ B_4\ B_5\ A_1\ A_2\ A_3\ A_4\ A_5$
C	$A_1\ B_1\ A_2\ B_2\ A_3\ B_3\ A_4\ B_4\ A_5\ B_5$
C	$B_1\ B_2\ A_1\ B_3\ B_4\ A_2\ B_5\ A_3\ A_4\ A_5$
C	$\cdots$

Las sentencias atómicas se ordenan en función del instante en el que acaban (que es cuando tienen efecto)

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, como 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.

Independencia del entorno de ejecución

El entrelazamiento preserva la consistencia

El resultado de una instrucción individual sobre un dato no depende de las circunstancias de la ejecución.

Supongamos que un programa P se compone de dos instrucciones atómicas, I_0 e I_1, que se ejecutan concurrentemente, (P es $I_0 \parallel I_1$), entonces:

- Si I_0 e I_1 no acceden a la misma celda de memoria o registro, el orden de ejecución no afecta al resultado final.
- Si $I_0 \equiv M \leftarrow 1$ y $I_1 \equiv M \leftarrow 2$, la única suposición razonable es que el resultado sea consistente. Por tanto, al final $M = 1$ ó $M = 2$, pero nunca por ejemplo $M = 3$.

En caso contrario, sería imposible razonar acerca de la corrección de los programas concurrentes.

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 más lento que una CPU pero el programa no debería asumir eso en el diseño del 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.

Estados e historias de ejecución de un programa concurrente

Estado de un programa concurrente

Valores de las variables del programa en un momento dado. Incluyen variables declaradas explícitamente y variables con información de estado oculta (contador del programa, registros, ...).

Un programa concurrente comienza su ejecución en un estado inicial y los procesos van modificando el estado conforme se van ejecutando sus sentencias atómicas (producen transiciones entre dos estados de forma indivisible).

Historia o traza de un programa concurrente

Secuencia de estados $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$, producida por una secuencia concreta de interfoliación.

Notaciones para expresar ejecución concurrente

- **Propuestas iniciales**: no separan la definición de los procesos de su sincronización.
- **Propuestas posteriores**: separan conceptos e imponen estructura.
- **Declaración de procesos**: rutinas específicas de programación concurrente $\Longrightarrow$ Estructura del programa concurrente más clara.

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).

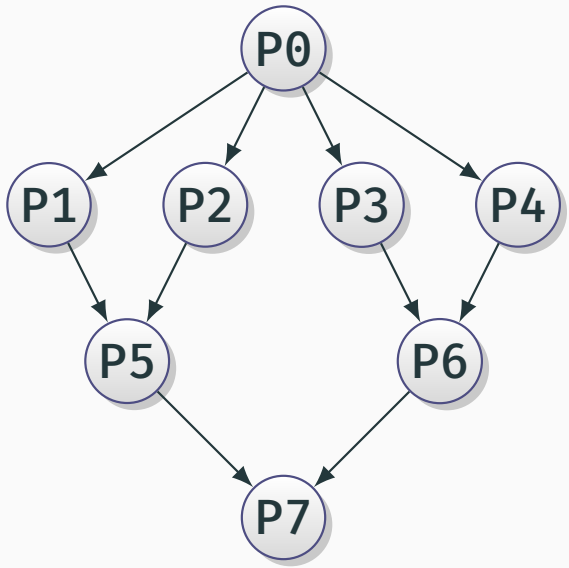
Sistemas Dinámicos

- 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

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

- Dadas dos actividades, A y B, una arista (flecha) desde A hacia B significa que B no puede comenzar su ejecución hasta que A haya finalizado.
- Muestra las restricciones de precedencia que determinan cuándo una actividad puede empezar en un programa.

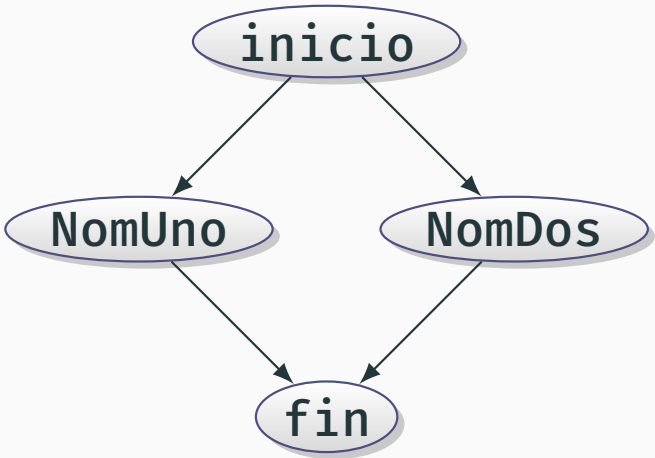


Definición estática de procesos

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 iniciaizan antes de comenzar ningún proceso.

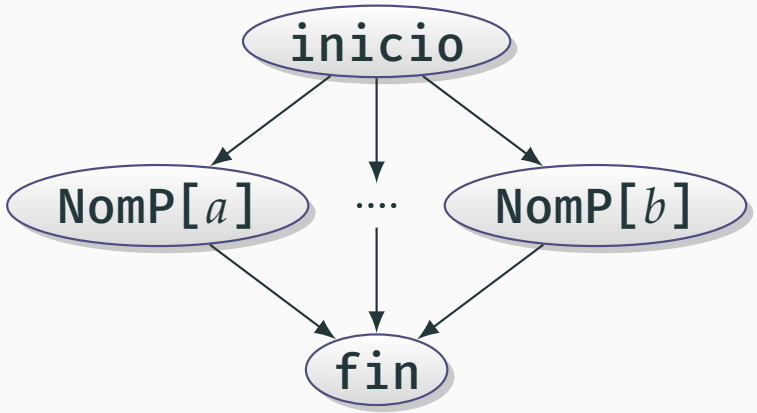
Definición estática de vectores de procesos

Se pueden usar definiciones estáticas de grupos de procesos similares que solo se diferencia 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 }
```



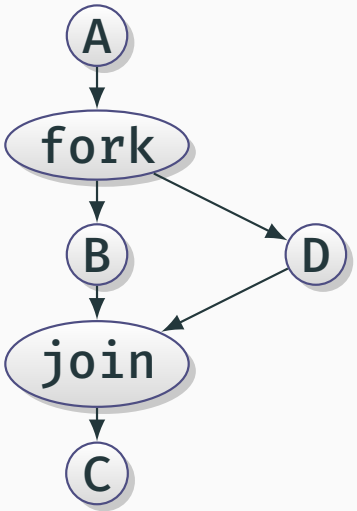
- En cada caso, a y b se traducen por dos constantes concretas (el valor de a será típicamente 0 o 1).
- El número total de procesos será $b-a+1$ (se supone que $a <= b$)

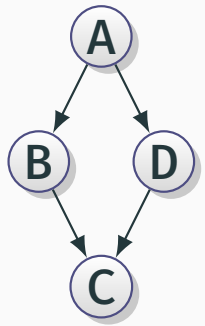
Creación de procesos no estructurada con fork-join.

- **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
```





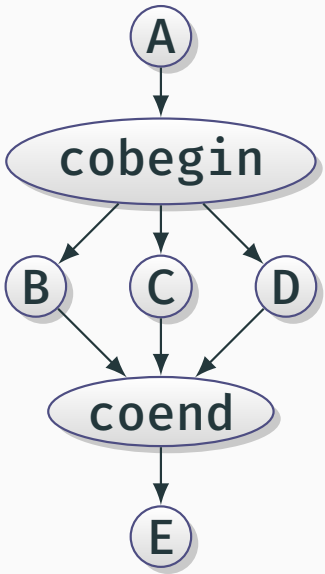
- **Ventajas:** práctica y potente, creación dinámica.
- **Inconvenientes:** no estructuración, difícil comprensión de los programas.

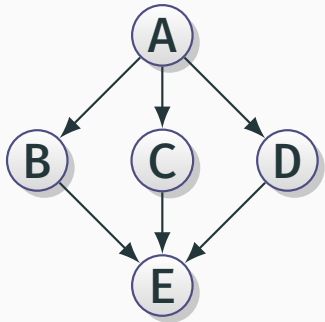
Creación de procesos estructurada con cobegin-coend

Las sentencias en un bloque delimitado por **cobegin-coend** comienzan su ejecución todas ellas a la vez:

- en el **coend** se espera a que se terminen todas las sentencias.
- Hace explícito qué rutinas van a ejecutarse concurrentemente.

```
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**.

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 procesos concurrentes ejecutan sus instrucciones atómicas de forma que, en principio, es completamente arbitrario el entremezclado en el tiempo de sus respectivas secuencias de instrucciones. Sin embargo, en un conjunto de procesos que **no son independientes entre sí** (es decir, son **cooperativos**), algunas de las posibles formas de combinar las secuencias no son válidas.

- En general, se dice que hay una **condición de sincronización** cuando esto ocurre, es decir, que hay alguna restricción sobre el orden en el que se pueden mezclar las instrucciones de distintos procesos.
- Un caso particular es la **exclusión mutua**, 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 tambien.

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

El ejemplo típico de EM ocurre en procesos con memoria compartida que acceden para leer y modificar variables o estructuras de datos comunes usando operaciones no atómicas (es decir, compuestas de varias instrucciones máquina o elementales que pueden solaparse con otras secuencias), aunque hay muchos otros ejemplos:

- Envío de datos a dispositivos que no se pueden compartir (p.ej., el bucle que envía una secuencia de datos que forma un texto a una impresora o cualquier otro dispositivo de salida vía el bus del sistema).
- Recepción de datos desde dispositivos (p.ej., un bucle que lee una secuencia de pulsaciones de teclas desde el teclado, también a través del bus).

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 esta 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

Si consideramos la instrucción `x:=x+1` (que forma la SC), veremos que una traducción típica a código máquina tendría estas tres instrucciones:

1.	`load` $\mathtt{r}_i \leftarrow \mathtt{x}$	cargar el valor de la variable `x` en un registro `r` de la CPU (por el proceso número i).
2.	`add` $\mathtt{r}_i$`,1`	incrementar en una unidad el valor del registro `r`
3.	`store` $\mathtt{r}_i \rightarrow \mathtt{x}$	guardar el valor del registro `r` en la posición de memoria de la variable `x`.

- Dos procesos (P_0 y P_1) ejecutan la asignación: la interfoliación es arbitraria.
- Cada proceso tiene su registro (los llamaremos $\mathtt{r}_0$ y $\mathtt{r}_1$).
- Ambos comparten `x`, cuyos accesos vía `load` y `store` son atómicos.

Posibles secuencias de instrucciones

Suponemos que inicialmente x vale 0 y ambos procesos ejecutan la asignación, puede haber varias secuencias de interfoliación, aquí vemos dos:

P_0	P_1	x
`load` $r_0 \leftarrow$ x		0
`add` r_0,1		0
`store` $r_0 \rightarrow$ x		1
	`load` $r_1 \leftarrow$ x	1
	`add` r_1,1	1
	`store` $r_1 \rightarrow$ x	2

P_0	P_1	x
`load` $r_0 \leftarrow$ x		0
	`load` $r_1 \leftarrow$ x	0
`add` r_0,1		0
	`add` r_1,1	0
`store` $r_0 \rightarrow$ x		1
	`store` $r_1 \rightarrow$ x	1

Partiendo de x$==0$, al final x puede valer 1 o 2, dependiendo de la interfoliación.

Instrucciones compuestas atómicas

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

```
{ 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.

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 disferentes.

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.