TEMA 1: INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE

1. Conceptos básicos y Motivación
2. Modelo Abstracto y Consideraciones sobre el Hardware
3. Notaciones para expresar ejecución concurrente
4. Exclusión mutua y Sincronización
5. Propiedades de sistemas concurrentes. Nociones de verificación

1. Conceptos básicos y Motivación
Conceptos basicos relacionados con la concurrencia

- **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****.*

  - Independiente de la implementación del paralelismo.
  - Es una abstracción

1. Conceptos básicos y Motivación
Programación Paralela, distribuida y de tiempo real

- **Programación paralela**

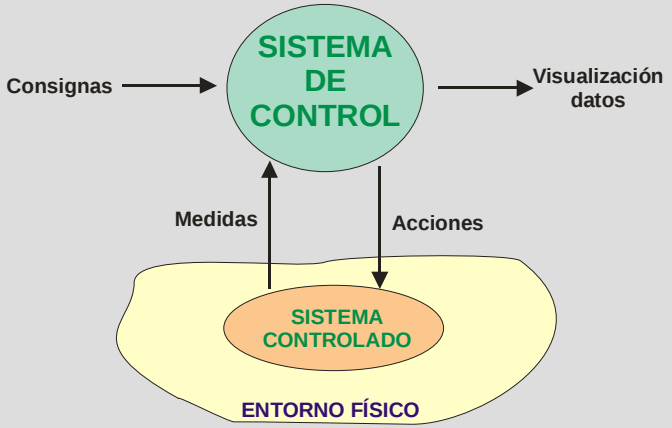
  Principal objetivo: acelerar resolución de problemas concretos aprovechando capacidad hardware paralelo.

- **Programación distribuida**

  Principal objetivo: hacer que varios componentes software localizados en diferentes ordenadores trabajen juntos.

- **Programación de tiempo real**

  Programación de sistemas **reactivos** (funcionan continuamente, recibiendo entradas y enviando salidas a/desde componentes hardware) que trabajan bajo restricciones de respuesta temporal muy estrictas **(sistemas de tiempo real**).



1. Conceptos básicos y Motivación
Motivación

Programación concurrente es más compleja que la progr. secuencial.

¿Por qué es necesario usar la Programación Concurrente?

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

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

1. Conceptos básicos y Motivación
Mejora de la eficiencia

- **Mejora el aprovechamiento de los recursos hardware existentes.**

- **Sistemas monoprocesador**:
  - Cuando la tarea que tiene el control dela CPU necesita realizar una E/S cede el control a otra, evitando espera ociosa CPU.
  - Permite que varios usuarios usen el sistema de forma interactiva (actuales *sistemas multiusuario*).

- **Sistemas multiprocesador** (multicore incluidos):
  - Reparto de las tareas entre los procesadores permite reducir tiempo de ejecución (**procesamiento paralelo**).
  - Aceleración aplicaciones costosas computacionalmente.

1. Conceptos básicos y Motivación
Mejora de la calidad

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

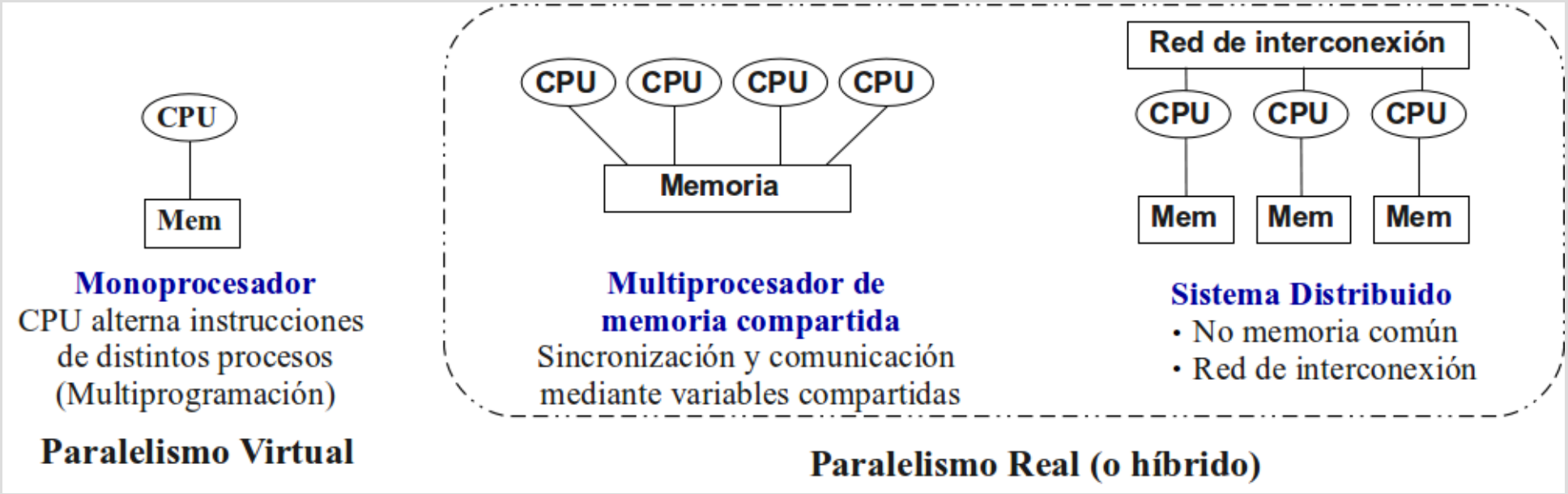
Ejemplos:

- **Servidor web para reserva de vuelos**: Más natural, considerar cada petición de usuario como un proceso e implementar políticas para evitar situaciones conflictivas (superar el límite de reservas en un vuelo).

- **Simulador del comportamiento de una gasolinera**: Más sencillo considerar surtidores, clientes, vehículos y empleados como procesos que cambian de estado al participar en actividades comunes, que considerarlos entidades de un único programa secuencial.

2. Modelo Abstracto y Consideraciones sobre el Hardware

Modelos de arquitecturas

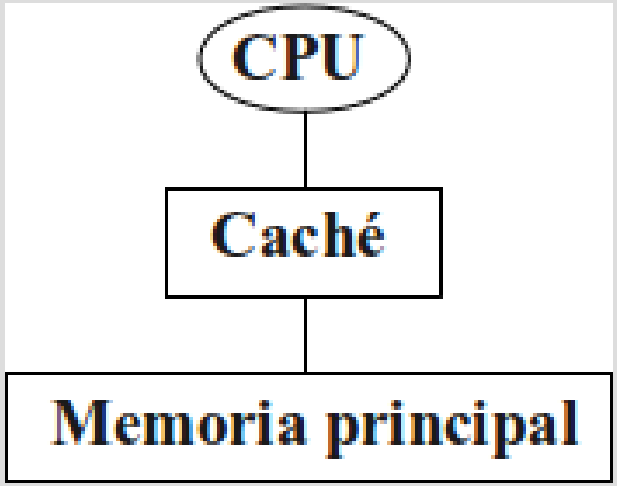


Mecanismos de implementación de la concurrencia

- 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**.

2. Modelo Abstracto y Consideraciones sobre el Hardware

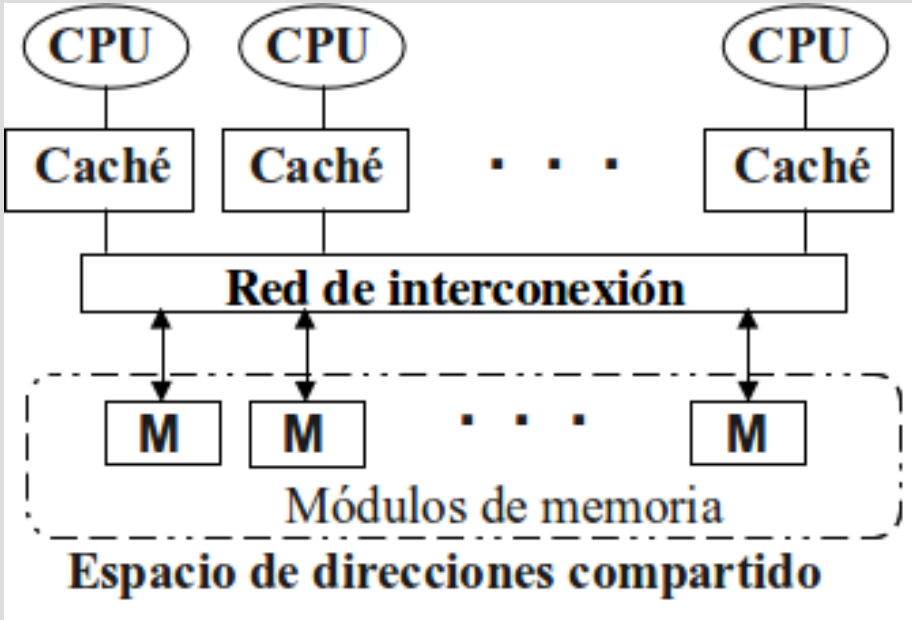
Concurrencia en Sistemas Monoprocesador



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

2. Modelo Abstracto y Consideraciones sobre el Hardware

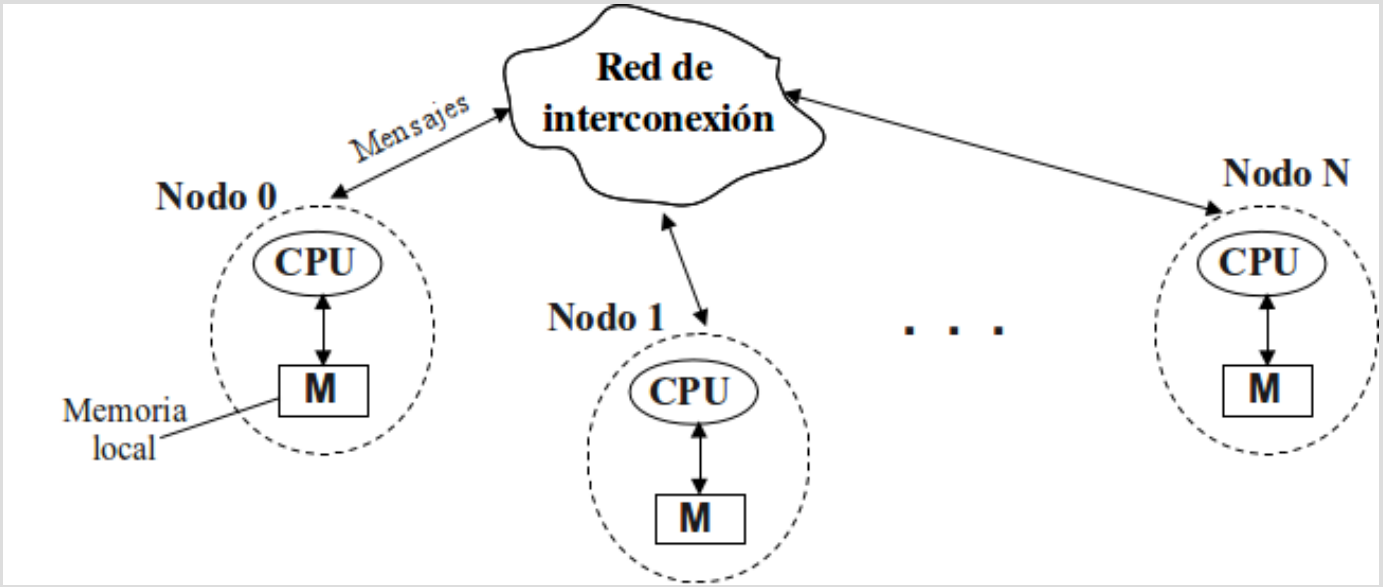
Concurrencia en Multiprocesadores de mem. compartida



- 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.

2. Modelo Abstracto y Consideraciones sobre el Hardware

Concurrencia en Sistemas Distribuidos



- **Sin memoria común**: cada procesador → espacio de direcciones privado.
- **Paso de mensajes:** procesadores interactúan transfiriéndose datos a través de una red de interconexión.
- **Programación distribuida**: además concurrencia, trata otros problemas: tratamiento fallos, transparencia, heterogeneidad, etc.
- **Ejemplos**: Clusters de ordenadores, internet, intranet.

2. Modelo Abstracto y Consideraciones sobre el Hardware

Sentencia Atómica

Sentencia atómica (indivisible) de un programa concurrente

Sentencia o instrucción de un proceso que siempre se ejecuta de principio a fin sin verse afectado su efecto por otras sentencias en ejecución de otros procesos del programa.

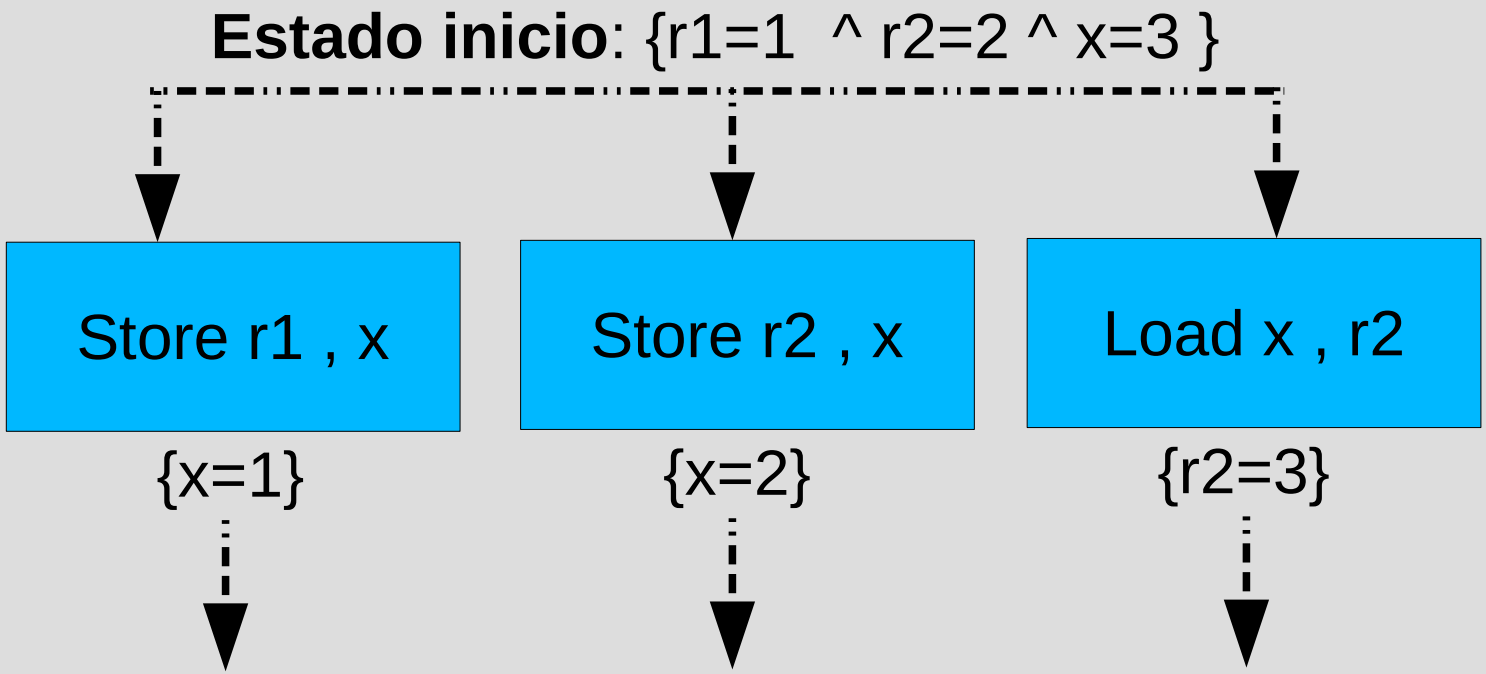
- **Efecto** en estado de ejecución cuando acaba instrucción **no dependerá** de cómo se estén ejecutando **otras instrucciones**.
- **Estado de ejecución** de un programa concurrente = {Valores de variables y registros de todos los procesos}.

- **Ejemplos de instrucciones atómicas**: instrucciones máquina del repertorio de un procesador:
  - **Load X, R** **Add R, v** **Store R, X**

2. Modelo Abstracto y Consideraciones sobre el Hardware

Ejemplo de Sentencia Atómica

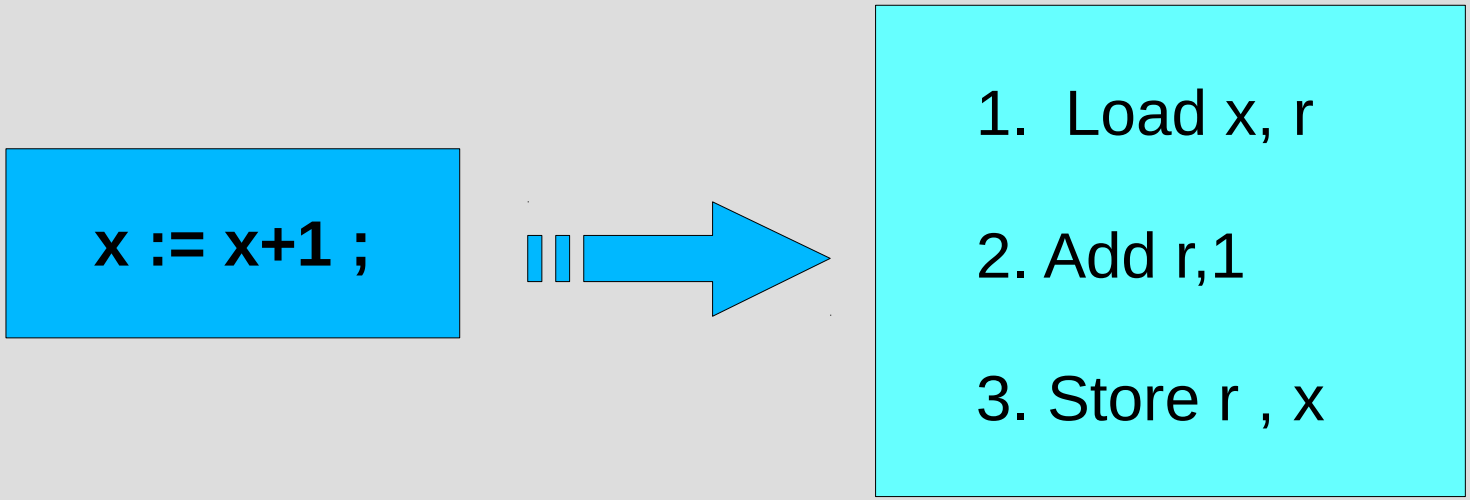
En una sentencia atómica, a partir del estado al inicio, el estado **justo al acabar** la instrucción está **determinado** como función del estado al inicio.



2. Modelo Abstracto y Consideraciones sobre el Hardware

Sentencias NO Atómicas

La mayoría de las **sentencias en lenguajes de alto nivel** son típicamente no atómicas



El valor de x justo al acabar depende de que haya o no otras sentencias ejecutándose y escribiendo simultáneamente sobre x.

Indeterminación: no se puede predecir estado final desde inicial.

2. Modelo Abstracto y Consideraciones sobre el Hardware

Modelo basado en la Interfoliación de sentencias atómicas

Sea **C** un **programa concurrente:** **C = P_A || P_B**

- ***MODELO EJECUCIÓN CONCURRENTE*: La ejecución de C puede dar lugar a cualquiera de las posibles interfoliaciones de sentencias atómicas de P_A y P_B que mantenga consistencia secuencial.**

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	$\ldots$

Ordenación sentencias atómicas se establece en función del **instante en el que acaban** (cuando tienen efecto).

2. Modelo Abstracto y Consideraciones sobre el Hardware

Modelo de interfoliación como Abstracción

Modelo basado en estudio de las posibles secuencias de interfoliación → Abstracción

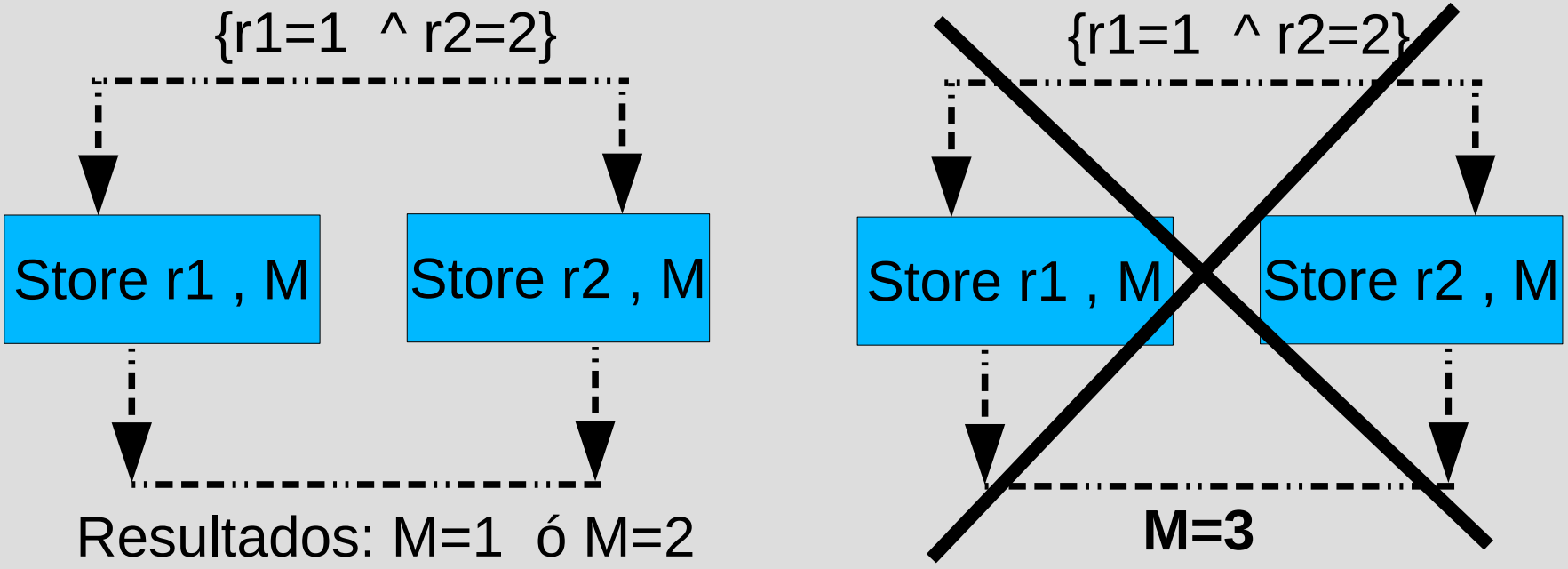
- Considera **solo características relevantes** que determinan resultado ejecución.
- **Simplifica análisis y diseño** programas concurrentes.
- **Ignora detalles no relevantes**:
  - Áreas de memoria asignadas procesos.
  - Registros usados.
  - Costo cambios de contexto.
  - Política asignación de CPU. de S.O.
  - etc.

2. Modelo Abstracto y Consideraciones sobre el Hardware

Independencia del entorno de ejecución

El entrelazamiento preserva la consistencia

- Resultado de instrucción individual sobre un dato no depende circunstancias ejecución.



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

2. Modelo Abstracto y Consideraciones sobre el Hardware

Hipótesis del Progreso Finito (1)

Hipótesis del 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 no se debería asumir eso al diseñar un programa concurrente.

- Si se hicieran suposiciones temporales:
  - Difícil detectar y corregir fallos.
  - Corrección dependería de configuración de ejecución, que puede cambiar.

2. Modelo Abstracto y Consideraciones sobre el Hardware

Hipótesis del Progreso Finito (2)

Progreso finito → Velocidad de ejecución cada proceso ≠ 0

2 Consecuencias:

- **Punto de vista global**

  Durante ejecución programa conc., en cualquier momento existirá al menos 1 proceso preparado (eventualmente se permitirá la ejecución de algún proceso).

- **Punto de vista local**

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

2. Modelo Abstracto y Consideraciones sobre el Hardware

Estados e Historias de ejecución

Estado de un programa concurrente

Valores de las variables del programa en un momento dado.

Tanto variables declaradas explícitamente como variables con información de estado oculta (contador del programa, registros, ...).

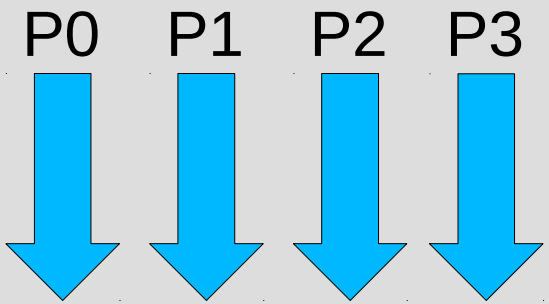
- Un programa concurrente comienza su ejecución en un estado inicial S0.
- Los procesos van modificando el estado conforme van ejecutando sus sentencias atómicas (producen transiciones entre dos estados).

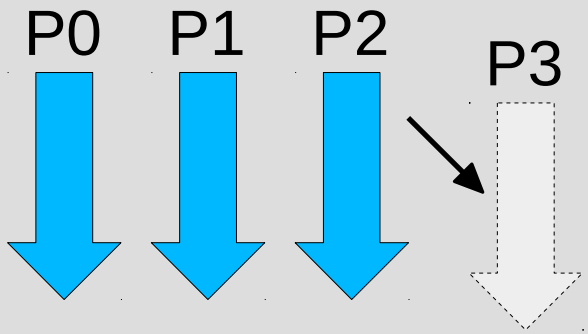
Historia o traza de un programa concurrente

Secuencia de estados **S0** $\rightarrow$ **S1** $\rightarrow$ **. . .** $\rightarrow$ **Sn** , producida por una secuencia concreta de interfoliación.

3. Notaciones para expresar ejecución concurrente

- **Propuestas iniciales**: no separan Definición procesos de su sincronización.
- **Propuestas posteriores**: separan ambos conceptos e imponen estructura.
- **Declaración de procesos**: rutinas específicas de programacón concurrente ⇒ Estructura del programa concurrente más clara.
- **Sistemas Estáticos:** Número de procesos fijado en el código fuente. Se activan al lanzar el programa.
  - Message Passing Interface (MPI-1).
- **Sistemas Dinámicos:** Procesos/hebras se pueden activar/desactivar en cualquier momento ejecución.
  - OpenMP, PThreads, MPI-2.





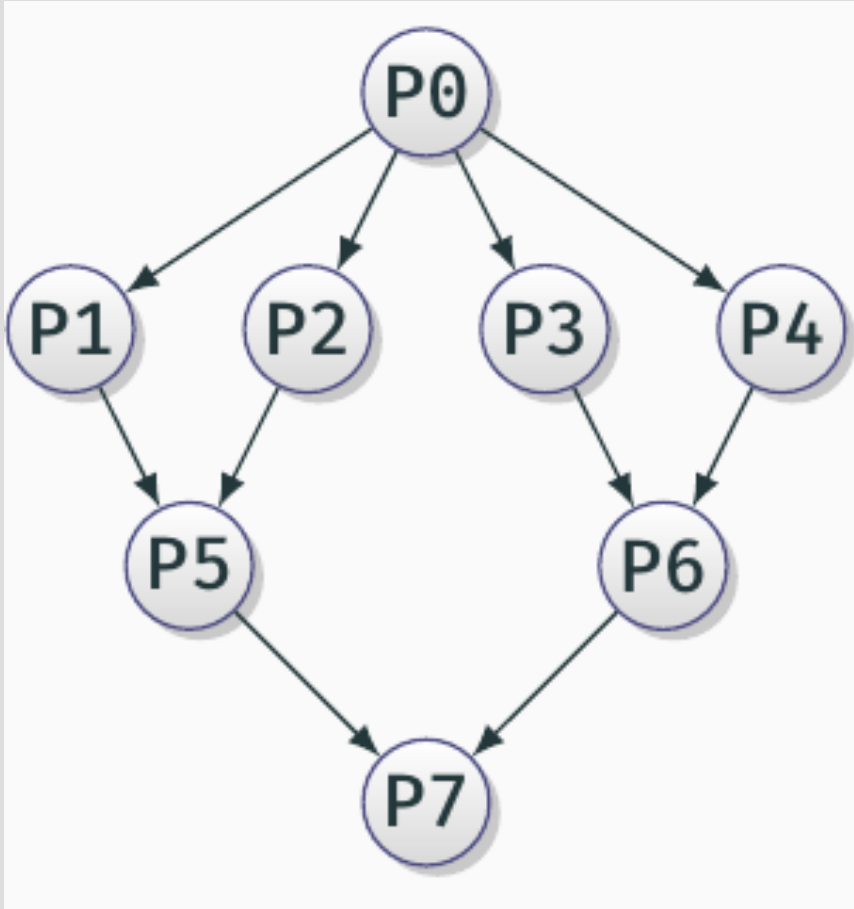
3. Notaciones para expresar ejecución concurrente

Grafo de Sincronización

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

- **Arista desde A hacia B**: B no puede comenzar su ejecución hasta que A haya finalizado.

- Muestra **restricciones de precedencia** entre actividades.



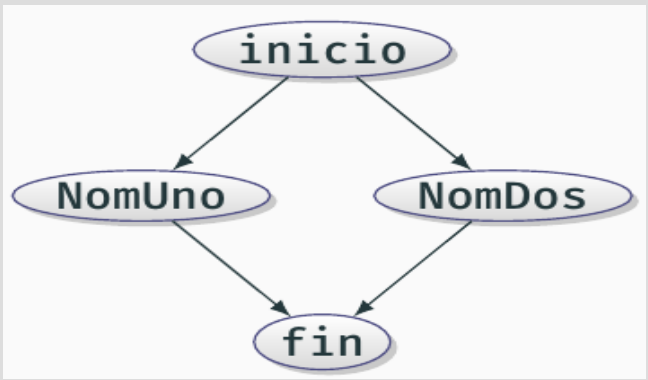
3. Notaciones para expresar ejecución concurrente

Definición estática de Procesos

Definición Procesos individuales

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

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

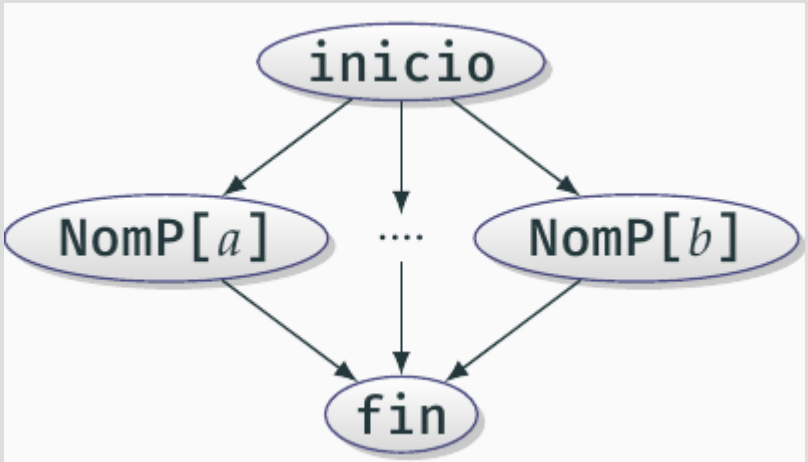


Vectores de Procesos

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

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

...   { otros procesos }
```



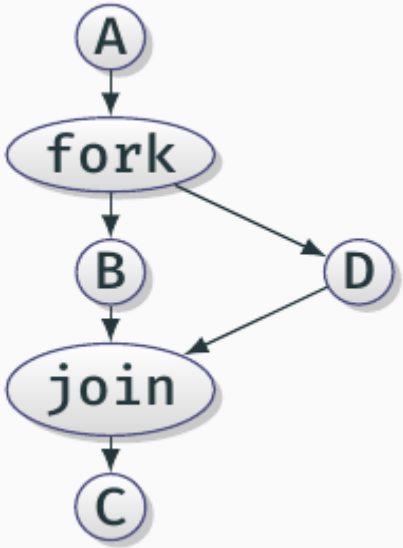
3. Notaciones para expresar ejecución concurrente

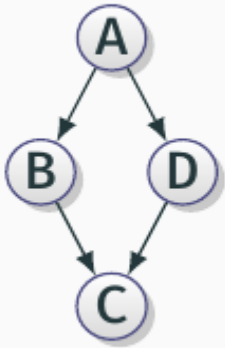
Creación no estructurada: fork-join

- **Fork A**: Especifica que la rutina **A** puede comenzar su ejecución, al mismo tiempo que comienza la sentencia siguiente (bifurcación).
- **Join B**: Espera la terminación de la rutina **B**, 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.

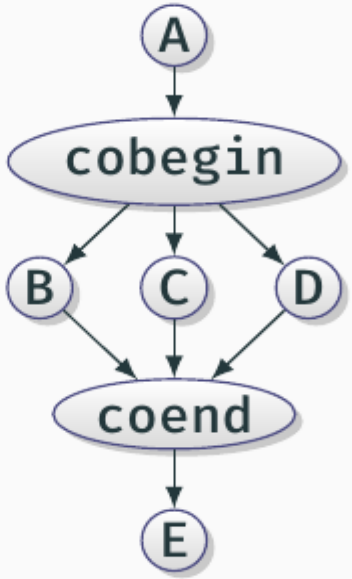
3. Notaciones para expresar ejecución concurrente

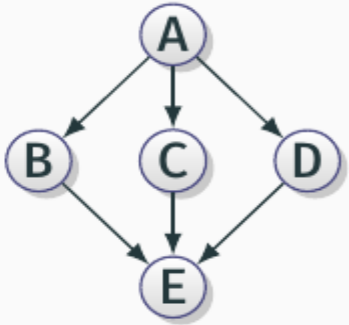
Creación estructurada: cobegin-coend

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

- En **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
```





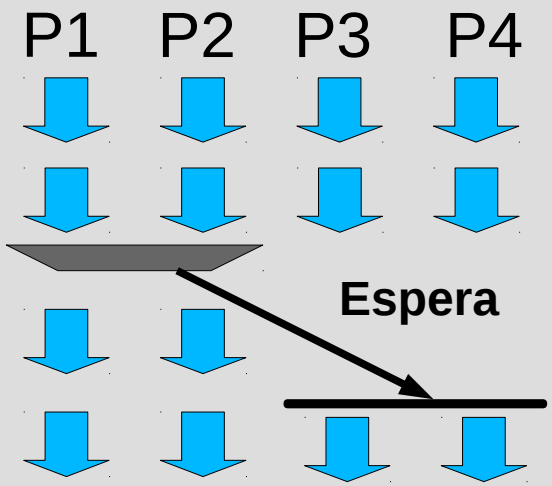
- **Ventaja**: impone estructura: 1 única entrada y 1 única salida ⇒ Más fácil de entender.
- **Inconveniente**: menor potencia expresiva que fork-join.

4. Exclusión mutua y Sincronización

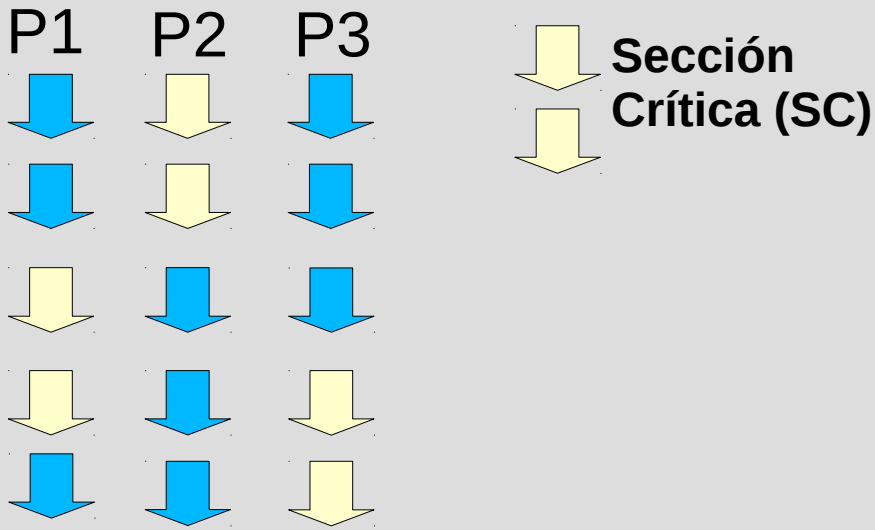
Modelo abstracto → **Entrelazamiento completamente arbitrario** de sus secuencias de instrucciones.

Programa Conc: **Algunos de los posibles entrelazamientos no son válidos**.

- **Condición de sincronización**: Restricción sobre el orden en el que se pueden mezclar las instrucciones de distintos procesos.



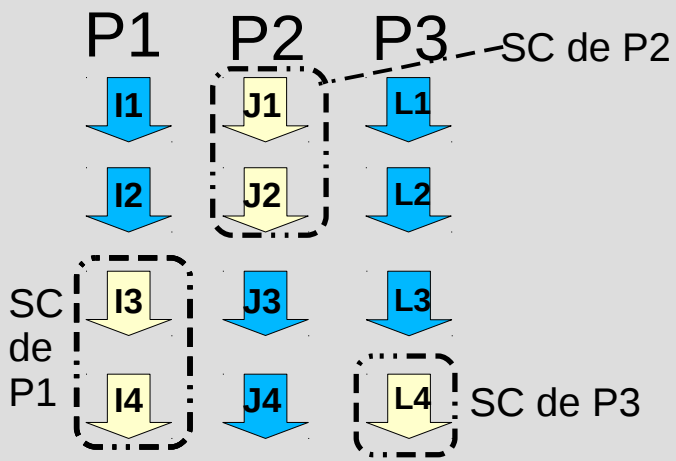
- **Exclusión mutua** (Caso particular): Secuencias finitas de instrucciones (SC) que deben ejecutarse de principio a fin por un único proceso, sin que otro proceso las esté ejecutando también a la vez.



4. Exclusión mutua y Sincronización
Exclusión mutua

Restricción de sincronización: Una o varias secuencias de instrucciones consecutivas del texto de uno o varios procesos.

- **Sección crítica (SC)**: Secuencias de instrucciones de diferentes procesos que no pueden entrelazar su ejecución.
- **Exclusión mutua (EM)**: En cada instante de tiempo, debe haber como mucho un proceso ejecutando cualquier instrucción de la sección crítica.
- Cada secuencia de instrucciones de la SC se ejecuta como mucho por un proceso de principio a fin, sin que simultáneamente otros procesos ejecuten ninguna instrucción de su SC.



Ejemplos de Secuencias:

- I1, J1, I2, J2, L1,L2, L3, J3, I3, I4, L4. CORRECTA

- I1, I2, I3, J1, … ERRÓNEA!

4. Exclusión mutua y Sincronización
Ejemplos de Exclusión mutua

1. Procesos con memoria compartida que acceden para leer y modificar variables o estructuras de datos comunes usando operaciones no atómicas (varias instrucciones máquina)

2. Envío de datos a dispositivos que no se pueden compartir (pantalla, …)

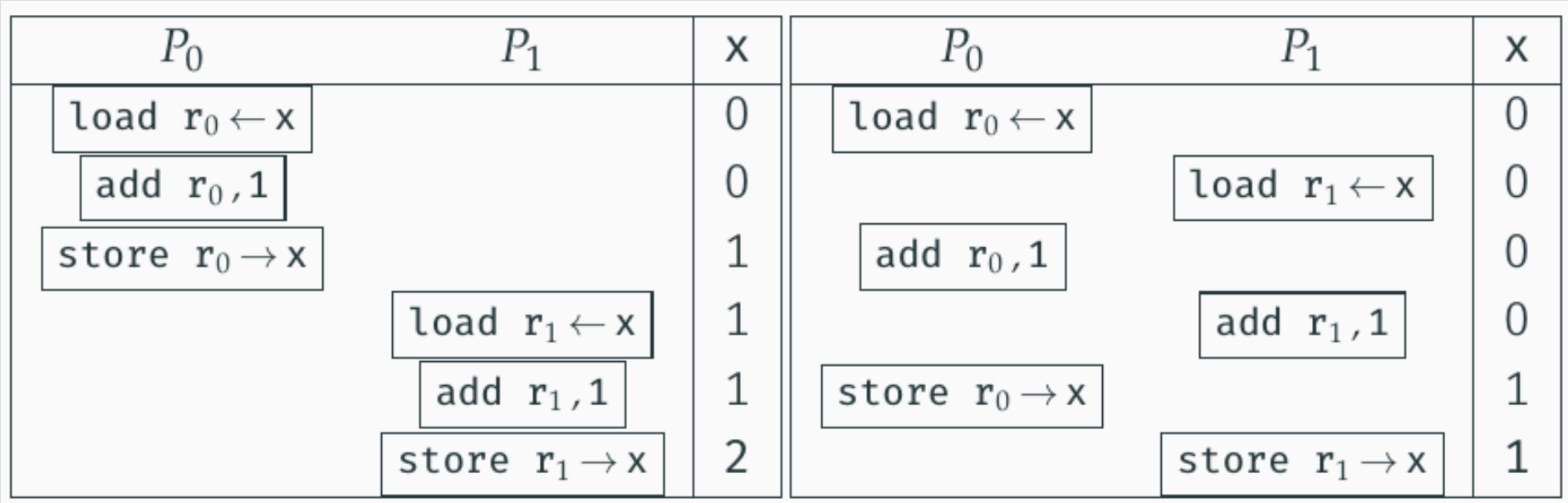
3. Recepción de datos desde dispositivos.

- **Ejemplo sencillo: Sección crítica** → Secuencias de instrucciones máquina que se derivan de operación incremento sobre variable entera (x) compartida **x:=x+1**.

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

4. Exclusión mutua y Sincronización
Ejemplo de Exclusión mutua: x:=x+1

- Suponemos que x=0 inicialmente
- Cada proceso (P_0 y P_1) tiene su propio registro (r_0 y r_1)
- Al final x podría valer 1 ó 2 si no se verifica la EM.



4. Exclusión mutua y Sincronización
Instrucciones compuestas atómicas

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

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

Al acabar, x puede tener un valor cualquiera del conjunto {−1, 0, 1 }

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

Aquí, x siempre finaliza con el valor 0

4. Exclusión mutua y Sincronización
Condición de Sincronización. Ejemplo

Condición de sincronización: no son correctas todas las posibles interfoliaciones de las secuencias de instrucciones atómicas de los procesos.

Típicamente, en un punto de ejecución, **uno o varios procesos deben esperar a que se cumpla una condición global** (depende de varios procesos)

Ejemplo: **Sincronización Productor-Consumidor**

```
{ 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
```

4. Exclusión mutua y Sincronización
Condición de Sincronización. Ejemplo

Los procesos solo funcionan como se espera si el **orden** en el que se entremezclan las **sentencias elementales** E y L es: **E, L, E, L, E, L, ...**

(1) Consumidor no lee hasta que Productor escriba nuevo valor en x (cada valor producido es usado una sola vez).

(2) Productor no escribe nuevo valor hasta que Consumidor lea el último almacenado en x (ningún valor producido se pierde).

- L, E, L, E, . . .incorrecta: lectura de valor indeterminado para x.
- E, L, E, E, L, . . . incorrecta: 2 escrituras sin lectura intermedia.
- E, L, L, E, L, . . . incorrecta: 2 lecturas del mismo valor.

```
{ 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
```

5. Propiedades Sistemas Concurrentes
Seguridad y Vivacidad

Propiedad de un programa concurrente: Atributo del programa que es cierto para todas las posibles secuencias de interfoliación (historias del programa).Hay **2 tipos**: **seguridad** (*safety*) y **vivacidad** (*liveness*).

Propiedades de Seguridad

Condiciones que **deben cumplirse Siempre**:

"N*unca pasará nada malo*"

- Requeridas en **especificaciones estáticas** del programa.
- **Fáciles de demostrar** y para cumplirlas se suelen restringir interfoliaciones.

Propiedades de Vivacidad

Condiciones que **deben cumplirse Eventualmente**:

"En algún momento *pasará algo bueno*"

- Requeridas en **especificaciones dinámicas** del programa.
- **Difíciles de demostrar**

5. Propiedades Sistemas Concurrentes
Ejemplos de propiedades

Ejemplos Prop. de Seguridad

(1) **Exclusión mutua**: 2 procesos nunca entrelazan ciertas subsecuencias de operaciones.

(2) **Ausencia Interbloqueo** (*Deadlock-freedom*): Nunca ocurrirá que los procesos se encuentren esperando algo que nunca sucederá.

(3) **Propiedad seguridad en Productor-Consumidor**: Consumidor debe consumir todos los datos producidos por Productor en el mismo orden.

Ejemplos Prop, de Vivacidad

(1) **Ausencia de inanición** (*Starvation-freedom*): Un proceso o grupo de procesos no puede ser indefinidamente pospuesto. En algún momento, podrá avanzar.

(2) **Equidad** (*Fairness*): 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.

5. Propiedades Sistemas Concurrentes
Verificación de programas concurrentes

¿ Cómo demostrar que un programa cumple una determinada propiedad ?

- **Posibilidad**: realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad.
- **Problema**: Sólo considera número limitado de historias ejecución y podría no mostrar 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
```

Varias historias llevan a x=1 ó x=2, pero podrían no darse.

5. Propiedades Sistemas Concurrentes
Enfoque operacional (análisis exhaustivo)

Enfoque operacional: Análisis exhaustivo de casos. Se chequea la corrección de todas las posibles historias.

- **Utilidad muy limitada** cuando se aplica a programas concurrentes complejos
  - **Número de interfoliaciones crece exponencialmente** con número de instrucciones.
- Para **programa P** (2 procesos, 3 sentencias atómicas por proceso) habría que estudiar **20 historias diferentes**.

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

5. Propiedades Sistemas Concurrentes
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**.
- **Sentencias atómicas** → **transformadores de predicados** (asertos).
- **Teoremas en la lógica:**

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

"*Si la ejecución de **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 Costo computacional**: Costo de la Prueba automática de corrección es **proporcional al número de sentencias atómicas** del programa.

5. Propiedades Sistemas Concurrentes
Invariante global

- **Invariante global**: Predicado que referencia variables globales
  - Cierto en el estado inicial de cada proceso
  - Se mantiene cierto ante cualquier asignación dentro procesos.

- **Ejemplo**: **Productor-Consumidor**, Invariante global sería:

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