Sistemas concurrentes y distribuidos. Resumen.

Índice

[1. Tema 1. Conceptos básicos 2](#_Toc83717595)

[1.2. Diferencia programación paralela, distribuida y de tiempo real 2](#_Toc83717596)

[1.3. Modelos de arquitecturas 3](#_Toc83717597)

[1.3.1. Paralelismo virtual 3](#_Toc83717598)

[1.3.2. Multiprocesador de memoria compartida 3](#_Toc83717599)

[1.3.3. Concurrencia en sistemas distribuidos 3](#_Toc83717600)

[1.4. Modelo abstracto de concurrencia 4](#_Toc83717601)

[1.4.1. Modelo de interfoliación como Abstracción 4](#_Toc83717602)

[1.4.2. Hipótesis del progreso finito 5](#_Toc83717603)

[1.4.3. Estados e historias de ejecución 5](#_Toc83717604)

[1.5. Notación para expresar ejecución concurrente 5](#_Toc83717605)

[1.5.1. Grafos de sincronización 5](#_Toc83717606)

[1.6. Exclusión mutua y sincronización 6](#_Toc83717607)

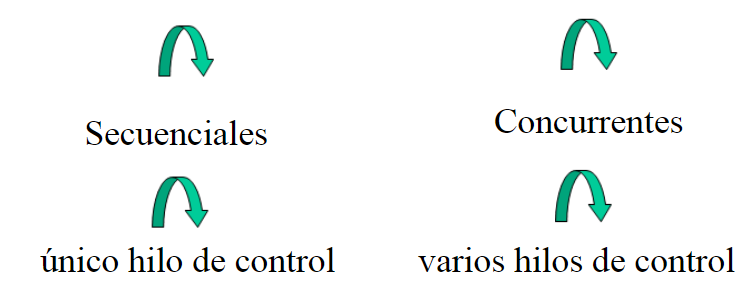
[1.7. Propiedades de un programa concurrente 7](#_Toc83717608)

[1.7.1. Comprobación de un programa concurrente 7](#_Toc83717609)

# Tema 1. Conceptos básicos

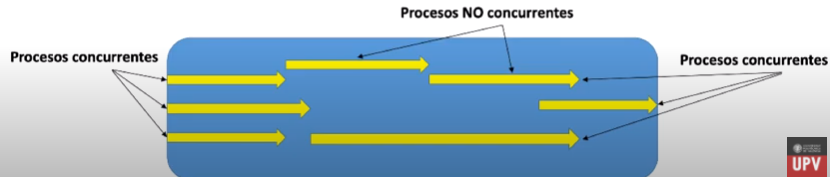
**Programa secuencial**: es un programa compuesto por declaraciones de datos e instrucciones sobre dichos datos que se ejecutan secuencialmente.

**Programa concurrente**: conjunto de procesos secuenciales que se ejecutan lógicamente (puede ser ejecutado en paralelo pero si los recursos hardware no lo permiten, se ejecuta secuencial) en paralelo.



**Proceso:** es la ejecución de un programa secuencial.

**Concurrencia:** es la capacidad que tiene un sistema para procesar más de un hilo (proceso) al mismo tiempo.



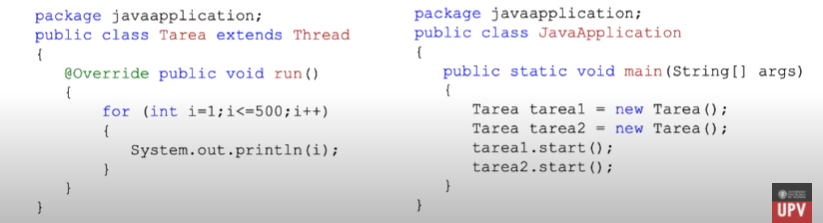
Los procesos NO concurrentes no se ejecutan a la misma vez en la línea del tiempo. Los otros sí. Puede haber concurrencia en un sistema monoprocesador. Hay concurrencia solo con que los procesos “convivan” en el mismo instante de tiempo.

Quiere decir: si se diese el caso de que solo hay un procesador, los procesos se van alternando el procesador. No se ejecutan a la vez, pero siguen siendo concurrentes.

Lo que expresa que se ejecuten al mismo tiempo es el paralelismo.

El paralelismo implica concurrencia: si se ejecutan a la misma vez, quiere decir que coexisten a la vez por lo que son concurrentes. Sólo pueden ser paralelos si hay mas de un procesador.

La concurrencia no implica paralelismo. Puede ser que coexistan a la vez o no.



Este código es concurrente, ya que se ejecutan dos hebras en el mismo momento. Se entrelazarán las salidas. No podemos saber si es paralelo, porque no sabemos si se lanzan en dos hilos distintos del procesador (no sabemos si se ejecutan al mismo tiempo).

La concurrencia implica que varios procesos o tareas coexistan en el mismo instante de tiempo (pero no necesariamente se ejecuten al mismo tiempo.

El paralelismo implica que dos procesos o más se ejecuten a la vez.

**Programación concurrente: o PC** es un conjunto de notaciones y técnicas de programación usadas para expresar paralelismo potencial y resolver problemas de sincronización y comunicación. Los beneficios de la PC son la mejora en la **eficiencia** y en la **calidad**:

Mejora de la eficiencia

Se mejora el aprovechamiento de los recursos hardware. En los sistemas monoprocesador, cuando una tarea necesita una E/S cede el procesador a otra tarea. En los multiprocesador se reparte las tareas entre los procesadores (**procesamiento paralelo**).

Mejora de la calidad

A veces es mejor desarrollar un mismo programa en términos de varios procesos secuenciales ejecutándose concurrentemente.

Las técnicas de programación concurrente son independientes de cómo es el posible paralelismo. Es una abstracción del hardware.

## Diferencia programación paralela, distribuida y de tiempo real

**Paralela:** su objetivo es acelerar la resolución de problemas concretos mediante el aprovechamiento de la capacidad de procesamiento en paralelo del hardware disponible. Se utiliza para cálculos complejos, por ejemplo.

**Programación distribuida:** Varios componentes software en pcs distintos trabajen juntos. Ordenadores que no comparten una zona de memoria común.

**Programación de tiempo real:** se centra en la programación de sistemas que funcionan continuamente con unas restricciones muy estrictas en cuanto al tiempo de respuesta.

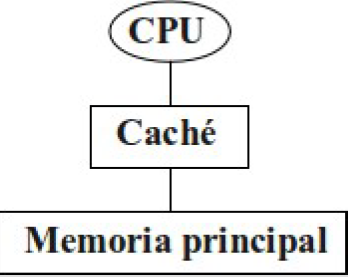
## Modelos de arquitecturas

Dependen de la arquitectura.

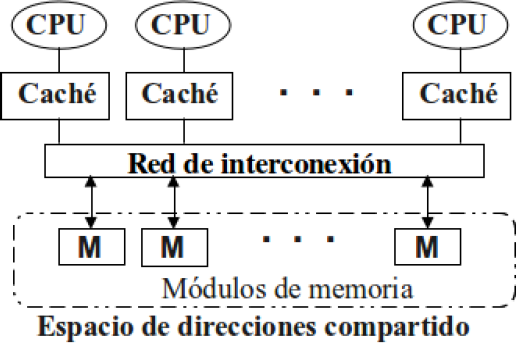
El tipo de paralelismo afecta a la eficiencia pero no a la corrección.

Un problema es correcto independientemente si es paralelo o no.

### Paralelismo virtual

También llamado concurrencia en sistemas monoprocesador, ya que no existe más que un procesador. El SO gestiona cómo se reparten los ciclos de CPU entre los procesos. Se aprovecha más la CPU, habilita un servicio interactivo a varios usuarios, permite soluciones de diseño concurrentes y la sincronización y comunicación se hace mediante **variables compartidas (un único procesador que accede a una única zona de memoria). En teoría, se podrían ejecutar en paralelo.**

### Multiprocesador de memoria compartida

Los procesadores pueden compartir o no físicamente la misma memoria pero comparte un **espacio de direcciones.** La interacción entre los procesos se implementa mediante **variables compartidas.**

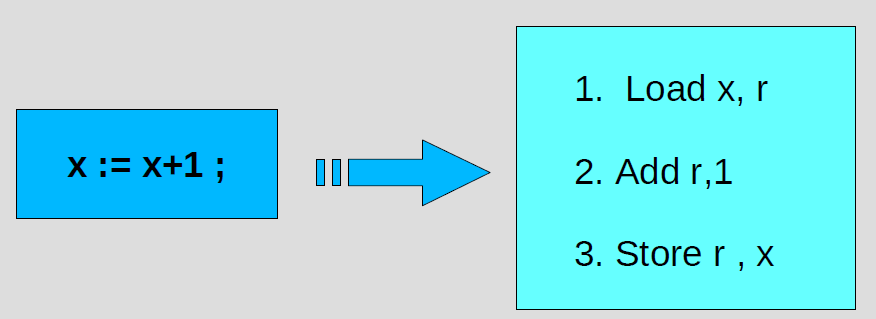
### Concurrencia en sistemas distribuidos

No se comparte memoria. Cada procesador tiene su memoria. La sincronización se realiza mediante **paso de mensajes.**

## Modelo abstracto de concurrencia

Una **sentencia atómica** de un programa concurrente es una sentencia o instrucción de un proceso que siempre se ejecuta de principio a fin sin verse afectado por otras sentencias en ejecución de otros procesos del programa. El efecto en el estado no varía (valores de variables y registros de todos los procesos). No depende nunca de otras instrucciones concurrentes. Quiere decir que el estado al finalizar está **determinado**.

Este es un ejemplo de sentencia no atómica:



Primero se carga el valor de X en el registro r, luego se le suma 1 y al final se coloca el resultado de r a la variable X. Por lo tanto, si hay otros procesos que utilicen X, el resultado no es predecible. **No es atómica.**

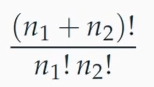
El estado final está determinado por el estado de inicio.

### Modelo de interfoliación como Abstracción

Es un modelo que se basa en el estudio de las posibles secuencias de interfoliación de las sentencias. Este modelo sólo trabaja con las características relevantes que puedan causar cambios en el resultado. Ignora detalles como:

* Áreas de memoria asignadas a procesos
* Registros usados
* Costo del cambio de contexto
* Política de asignación

Para calcular el número posible de interfoliaciones, utilizamos:



Binomial para 2 procesos o multinomial para 3 o más.

**El entrelazado de las sentencias atómicas preserva la consistencia.** Quiere decir que el resultado de una instrucción no depende de las otras instrucciones. Las instrucciones **no acceden a la misma celda de memoria.**

### Hipótesis del progreso finito

No se puede hacer suposiciones de tiempos de ejecución de procesos, salvo que es mayor que 0.

### Estados e historias de ejecución

Estado de programa concurrente 🡪 valores de las variables tanto las declaradas explícitamente como las variables de información de estado ocultas (contador de programa, registros, puntero de pila). Los procesos van modificando el estado conforme ejecutan sentencias atómicas. Pasan de un estado a otro.

**Consistencia secuencial estricta**

Una instrucción atómica de lectura siempre leerá el valor escrito por la inmediata anterior atómica de escritura.

Historia o traza de programa concurrente 🡪 Secuencia de estados producida por una interfoliación concreta.

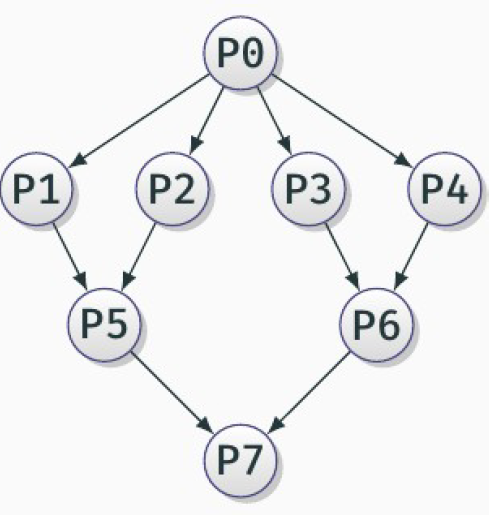
## Notación para expresar ejecución concurrente

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

* Sistemas estáticos: número de procesos fijados en código fuente y se activan al lanzar dicho programa. (MPI)
* Sistemas dinámicos: Procesos/hebras se pueden activar/desactivar en tiempo de ejecución. (MPI-2 o Threads).

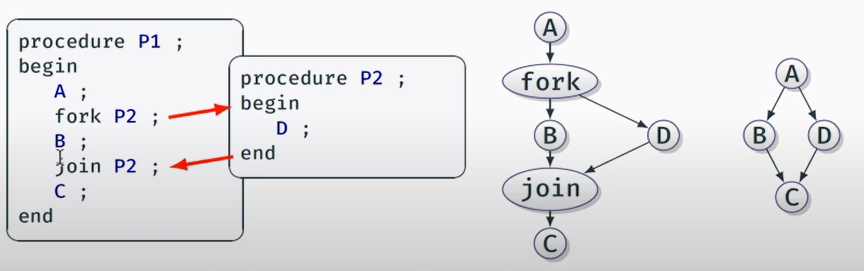
### Grafos de sincronización

Se trata de grafos dirigidos acíclicos donde cada nodo representa una secuencia de sentencias del programa (o una actividad).

Un nodo no puede comenzar hasta que termine el padre. Muestra **restricciones de precedencia** entre actividades.

**FORK-JOIN**

Fork nos indica que existe una bifurcación y JOIN nos indica que debe esperar a que finalice. Así, en el ejemplo siguiente cuando se produce un FORK tras A quiere decir que se ejecuta concurrentemente D. Se ejecuta B y aparece un JOIN, que nos indica que debe esperar la finalización de D (la bifurcación) para ejecutar C.



Ventajas: práctico y potente.

Desventajas: no estructurado (fork y join no se anidan uno dentro de otro. Sólo se puede hacer JOIN de un subconjunto de bifurcaciones previas FORK) y difícil comprensión.

**COBEGIN-COEND**

Se trata de una creación **estructurada.** Las sentencias entre un cobegin-coend empiezan todas a la vez. COEND indica que tiene que acabar todas las sentencias que se han ejecutado concurrentemente.

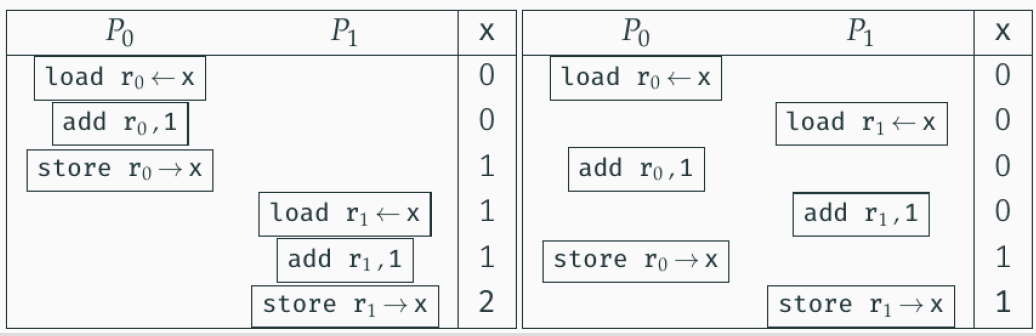
Ventajas: impone estructuras y es más fácil de entender

Desventajas: menor potencia expresiva.

## Exclusión mutua y sincronización

Para ciertas sincronizaciones, debemos restringir algunas interfoliaciones para cumplir. A esto se le llama **condición de sincronización**.

La **exclusión mutua** es una secuencia finita de instrucciones (**sección crítica)** que deben ejecutarse de principio a fin por un **único proceso** sin que otro proceso las esté ejecutando a la vez.



Las secciones críticas nunca pueden entrelazarse entre sí.

SECCIÓN CRÍTICA

Es una secuencia de instrucciones de diferentes procesos que no pueden entrelazar su ejecución. Por ejemplo:

X = X + 1

## Propiedades de un programa concurrente

Atributos del programa que es cierto para todas las posibles secuencias de interfoliación (historias del programa). Existen dos tipos:

* **Propiedad de seguridad:** Condiciones que deben cumplirse siempre. “*Nunca pasará nada malo”.* El programa nunca debe ejecutar algo que no se pueda.Se requieren en especificaciones estáticas y son fáciles de probar. Por ejemplo, la exclusión mutua (se restringen interfoliaciones con la condición de sincronización), ausencia de interbloqueo.

Es decir: los programas concurrentes se ejecutan independientemente de si son seguros o no. Esto quiere decir que las hebras se van a ejecutar. El programa es seguro cuando el resultado obtenido es el esperado.

EJEMPLO DE PAÑUELO: siempre habrá una persona que salga a correr cuando se diga un número, siempre ganará uno cada vez que se dice un número, nunca saldrán dos personas al mismo número, etc

* **Propiedad de vivacidad:** Sólo deben cumplirse eventualmente. “*En algún momento pasará algo bueno”*. Son cosas que ocurren y son aceptables pero que no pueden ocurrir siempre. Se requieren en las especificaciones dinámicas y son difíciles de demostrar. Por ejemplo, la ausencia de inanición (un proceso puede ser pospuesto pero no puede ser pospuesto indefinidamente) y la equidad (un proceso que quiera progresar debe hacerlo con “justicia” con los demás procesos).

EJEMPLO DE PAÑUELO

### Comprobación de un programa concurrente

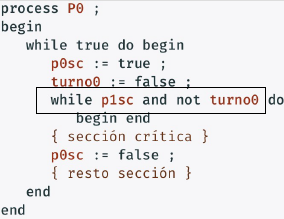
Existen dos métodos para comprobar la concurrencia de un programa:

* **Enfoque operacional:** Análisis exhaustivo de todos los casos. Esto no es útil porque habría que estudiar todas las historias del programa (todas las secuencias).
* **Enfoque axiomático:** se define un sistema lógico formal que permite establecer propiedades en base a axiomas y reglas. Se usan fórmulas lógicas. El costo computacional es proporcional al número de sentencias atómicas.
* **Invariante global:** Es un predicado que referencia variables globales. Inicia cierto y se mantiene cierto.

# Algoritmos y mecanismos de sincronización de memoria compartida

Vamos a ver soluciones para la exclusión mutua y la sincronización cuando un programa concurrente utiliza variables compartidas. Se distinguen entre:

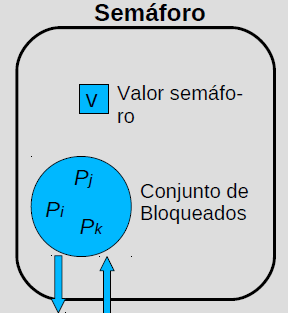
* **Soluciones de bajo nivel con espera ocupada** (bucles para realizar la espera). Existen dos tipos: soluciones software y soluciones hardware(cerrojos).



Estas soluciones tienen a fallar y consumen mucha CPU, ya que hay una espera ocupada.

* **Soluciones de alto nivel (se bloquean los procesos** en vez de espera ocupada). Algunos ejemplos son:

**Semáforos:** bloquean el proceso. No hay espera ocupada. Resuelven el problema de la exclusión mutua ya que impide que varios procesos ejecuten alguna sentencia de la sección crítica a la misma vez. Permiten resolver cualquier problema de sincronización pero puede llegar a ser complejo. La estructura de un semáforo es la siguiente:



Al inicio el conjunto de bloqueados estará vacío y se debe dar un valor inicial al semáforo. Tiene dos operaciones:

* Sem\_wait(semáforo): decrementa en 1 el semáforo. Si es 0, espera a que sea 1 para decrementarlo y avanzar.
* Sem\_signal(semáforo): aumenta en 1 el semáforo y desbloquea a un proceso del conjunto de bloqueados (si lo hay).

El problema de los semáforos es que impide el diseño modular (no hay una estructura como la de los monitores para separar en módulos).

**Monitores:** permiten definir objetos abstractos compartidos. Incluyen una colección de variables encapsuladas (recurso compartido) y un conjunto de procedimientos. Se accede en EM a las variables compartidas y se implementa sincronización mediante esperas bloqueadas. La exclusión mutua está garantizada por definición: solo un proceso puede ejecutar a la vez un procedimiento del monitor y los demás esperan en la cola del monitor o en las variables condición. Los componentes de un monitor son los siguientes:

* Variables permanentes: forma el estado interno del recurso compartido. Solo se accede a ellas desde dentro del monitor (con los procedimientos y con el constructor)
* Procedimientos: modifican el estado del monitor.
* Código de inicialización: fija el estado inicial del monitor.

EL monitor es un objeto pasivo. Funciona cuando se llaman a sus procedimientos. Además, tiene que tener una **política FIFO** para garantizar la **vivacidad.**

Las **variables condición** permiten a los procesos bloquearse en una cola cuando no cumplen ciertas condiciones. Se utiliza la operación **cond.wait()** para bloquear y meter en la cola y **cond.signal()**  para desbloquear de la cola. Además, reactiva el que lleve más tiempo (FIFO).

Así se evita la inanición.

## Tipos de monitores

Cuando se ejecuta un signal() sobre una variable condición, hay varias semánticas de monitor. Esto es porque si se ejecutan ambos, se viola la EM del monitor. Para ello, puede continuar el señalado, puede esperar, etc.

### Señalar y continuar

El señalado abandona la cola condición y se espera en la del monitor a pelear la EM. El señalador continua.

### Señalar y salir

El señalador abandona el monitor después del signal. El señalado reanuda inmediatamente la ejecución.

### Señalar y esperar

El señalador se bloquea en la cola del monitor para pelear EM tras el signal y el señalado se reanuda inmeditamente.

### Señalar y espera urgente

Al señalar, se espera en una cola de urgentes. En cuanto que el monitor está libre, entran primero los de la cola de urgentes.

Se podría implementar un monitor con semáforos de la siguiente manera:

* La cola del monitor con un semáforo binario, que marca 1 si no hay procesos ejecutando procedimientos del monitor y 0 si lo hay)
* Las colas condición con un contador para saber el número de procesos que hay y un semáforo definido a 0 para liberar procesos.
* La cola de urgentes lo mismo: un semáforo y una variable entera.

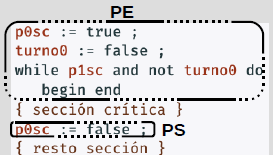
Los semáforos tienen la misma potencia expresiva que los monitores pero los monitores son más sencillos.

## Soluciones software con espera ocupada para EM

### Algoritmos para exclusión mutua

Vamos a ver dos algoritmos de EM. La estructura de los procesos en estos algoritmos:

* Protocolo de entrada **(PE):** son las instrucciones que hacen esperar cuando no se puede acceder a la sección crítica.
* Sección crítica (**SC):** instrucciones que solo pueden ejecutarse por un proceso a la vez.
* Protocolo de salida (**PS):** Permite que otros procesos sepan que este proceso ha terminado SC.
* Resto de sentencias **(RS):** el resto de las sentencias.



Por lo tanto, para que un algoritmo **sea correcto** debe coumplir:

* Exclusión mutua: solo puede acceder un proceso a la vez a la SC. No suficiente.
* Progreso: tras acceder al PE, un proceso podrá salir y ejecutar la SC. Si no sucede, podría provocarse un interbloqueo.
* Espera limitada: un proceso no podrá esperar indefinidamente en el PE. Hay que definir un tope para que pueda entrar. Las esperas siempre serán finitas.

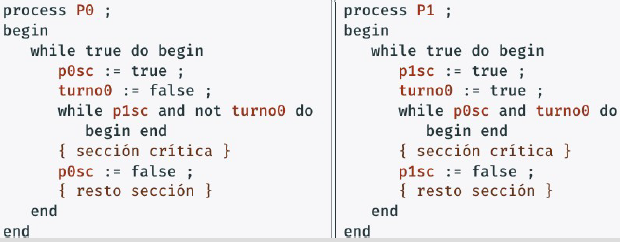
También tenemos propiedades deseables, como son la eficiencia o equidad.

### Refinamiento sucesido de Dijkstra

Comienza con una versión muy simple de un algoritmo incorrecto (no cumple alguna de las 3 propiedades mínimas: espera limitada, exclusión mutua y progreso) y lo va refinando hasta llegar a un algoritmo correcto. Este se llama **algoritmo de Dekker.**

### Algoritmo de Peterson

Es un algoritmo correcto pero más simple y eficiente que Dekker. Utiliza una variable turno para evitar el acceso simultaneo al PE y dos variables para PE.



## Soluciones hardware para EM

Se trata de meter a los procesos en bucles mediante **cerrojos.** Se pone el cerrojo cerrado al entrar en la SC y se abre al salir. Se basa en **espera ocupada.**  Son bucles sin hacer nada,repitiéndose una y otra vez.