# Teoría, conceptos importantes

# Diferencias entre "Signal and Continue" y "Signal and Wait" en la sincronización de monitores

Tanto "Signal and Continue" (SC) como "Signal and Wait" (SW) son disciplinas de señalización utilizadas en la programación concurrente con monitores para gestionar la interacción entre procesos que acceden a recursos compartidos. La principal diferencia radica en el comportamiento del proceso que envía la señal (signal) después de despertar a otro proceso que estaba esperando (wait) una condición.

**Signal and Continue (SC):**

En esta disciplina, después de que un proceso ejecuta signal(cv), **el proceso que envía la señal continúa su ejecución dentro del monitor**.

El proceso que estaba esperando en wait(cv) se coloca en la cola de entrada del monitor y competirá por el acceso exclusivo al monitor, junto con cualquier otro proceso que intente entrar.

**El proceso señalizador no cede el control del monitor inmediatamente**.

Esta disciplina **es no-preemptive (no-apropiativa)**.

**Signal and Wait (SW):**

En contraste con SC, después de que un proceso ejecuta signal(cv), **el proceso que envía la señal cede inmediatamente el control del monitor al proceso que estaba esperando**.

El proceso señalizador se coloca en la cola de entrada del monitor y esperará su turno para volver a entrar.

**El proceso que estaba esperando en wait(cv) obtiene acceso exclusivo al monitor y continúa su ejecución**.

Esta disciplina **es preemptive (apropiativa)**.

## Implicaciones de las diferentes disciplinas

La elección de la disciplina de señalización (SC o SW) puede afectar el comportamiento y la eficiencia de un programa concurrente:

**Orden de ejecución:** SC generalmente no garantiza un orden de ejecución específico entre el proceso señalizador y el proceso que estaba esperando. SW, por otro lado, garantiza que el proceso que estaba esperando se ejecutará inmediatamente después de la señal.

**Complejidad de la implementación:** SC suele ser más simple de implementar que SW, ya que no requiere un mecanismo para guardar el contexto del proceso señalizador y restaurarlo después de que el proceso que estaba esperando haya terminado.

**Rendimiento:** En algunos casos, SW puede ofrecer un mejor rendimiento que SC, ya que reduce la latencia al permitir que el proceso que estaba esperando se ejecute inmediatamente. Sin embargo, también puede aumentar la contención en la cola de entrada del monitor si muchos procesos envían señales con frecuencia.

## Ejemplos en las fuentes

Las fuentes proporcionan ejemplos concretos de la aplicación de SC y SW, así como sus ventajas e inconvenientes en diferentes escenarios:

**Fuente [Addison Wesley - Foundations Of Multithreaded Parallel And Distributed Programming.pdf]:** Esta fuente utiliza principalmente await statements, que

funcionan de manera similar a los monitores, e ilustra cómo se pueden implementar con semáforos. Si bien no se mencionan explícitamente SC o SW, el concepto de "passing the baton" ([1]) se asemeja a la disciplina SW, ya que un proceso cede el control a otro en un punto específico.

**Fuente [Addison Wesley - An Introduction to Parallel Computing 2nd Ed.pdf]:** Esta fuente se centra más en el paralelismo a nivel de memoria y comunicación entre procesos, y no entra en detalles sobre la implementación de monitores con SC o SW.

**Fuente [Clase-5---Programacin-Concurrente---2024.pdf]:** Esta fuente explica las diferencias entre SC y SW de manera clara y concisa ([2]), incluyendo diagramas de estado que ilustran el flujo de ejecución. Además, menciona que **en el contexto de la materia, se utilizará la disciplina Signal and Continue**.

### Elección de la disciplina adecuada

La elección entre SC y SW depende de los requisitos específicos del problema y del entorno de programación. No existe una respuesta única para determinar cuál es la mejor opción en todos los casos.

Es importante considerar factores como la necesidad de un orden de ejecución específico, la complejidad de la implementación, el rendimiento esperado y las posibles implicaciones para el starvation (inanición) de procesos.

# El problema de la sección crítica en programación concurrente

El problema de la sección crítica es un desafío fundamental en la programación concurrente, especialmente en sistemas donde múltiples procesos o hilos comparten recursos, como la memoria. Se refiere a la necesidad de garantizar que a lo sumo un proceso pueda acceder y modificar una sección crítica del código, o datos compartidos, a la vez. Si varios procesos intentan acceder a la sección crítica simultáneamente, esto puede llevar a condiciones de carrera y resultados impredecibles.

**Definición del problema de la sección crítica:**

El problema radica en diseñar protocolos de entrada y salida para la sección crítica que satisfagan las siguientes propiedades [1, 2]:

**Exclusión mutua:** Esta propiedad es fundamental, garantizando que solo un proceso pueda estar en la sección crítica en un momento dado [2]. Esto evita la interferencia entre procesos que acceden a los mismos datos compartidos.

**Ausencia de deadlock:** Si dos o más procesos intentan entrar en sus secciones críticas, al menos uno debe tener éxito eventualmente [2]. Un deadlock, donde ningún proceso puede avanzar, debe evitarse a toda costa.

**Ausencia de demora innecesaria:** Si un proceso desea acceder a la sección crítica y los demás procesos están en sus secciones no críticas o han terminado, el primer proceso no debe ser bloqueado para entrar [2].

**Entrada eventual (Eventual Entry):** Un proceso que intenta entrar en su sección crítica debe tener la oportunidad de hacerlo eventualmente [2]. Esto está relacionado con la justicia (fairness) en la programación concurrente, asegurando que ningún proceso se quede esperando indefinidamente.

**Soluciones al problema de la sección crítica:**

Las fuentes mencionan varias soluciones al problema de la sección crítica, cada una con sus ventajas y desventajas:

**Deshabilitar interrupciones [3]:** Es una solución simple para un único procesador donde las interrupciones se deshabilitan al entrar en la sección crítica y se habilitan al salir. Sin embargo, esta solución no es viable en sistemas multiprocesador.
**Solución de grano grueso [4, 5]:** Usa variables de bloqueo compartidas (in1, in2 en el ejemplo) para indicar si un proceso está en la sección crítica. Sin embargo, no garantiza la exclusión mutua en todas las situaciones (solo entre dos procesos) [5].
**Spin Locks [6, 7]:** Estos bloqueos utilizan instrucciones atómicas de bajo nivel como Test-and-Set (TS) para proteger la sección crítica. Los procesos compiten por el bloqueo en un bucle ocupado ("spinning") hasta que lo adquieren. Son eficientes en arquitecturas multiprocesador pero pueden llevar a una alta contención de memoria.
**Test-and-Test-and-Set (TTS) [8]:** Ofrece una mejora sobre el Test-and-Set simple al reducir la contención de memoria en algunos casos.
**Algoritmos Fair:**
**Tie-Breaker [9-11]:** Este algoritmo introduce un mecanismo de "desempate" para asegurar que cada proceso tenga la oportunidad de entrar en la sección crítica, evitando el starvation (inanición). Utiliza una variable adicional para rastrear el último proceso que comenzó su protocolo de entrada.
**Ticket [12-15]:** Similar a un sistema de tickets en el mundo real, los procesos toman un número y esperan su turno para entrar en la sección crítica. Es una solución justa y eficiente, pero requiere una instrucción Fetch-and-Add (FA) o una simulación de la misma.
**Bakery [15-18]:** Este algoritmo es más complejo pero no requiere instrucciones especiales como FA. Los procesos eligen un número (como en una panadería) y esperan a que sea su turno, basándose en una comparación de pares de números.

**Alternativas a los Spin Locks:**
Además de los spin locks, existen otros mecanismos de sincronización, como semáforos y monitores, que se utilizan para resolver el problema de la sección crítica de manera más estructurada y eficiente en muchos casos:

**Semáforos [19-21]:** Son variables enteras especiales que se utilizan para controlar el acceso a recursos compartidos. Proporcionan operaciones atómicas como P (esperar) y V (señalizar), que permiten la sincronización entre procesos.
**Monitores [22]:** Son estructuras de datos abstractas que encapsulan variables compartidas y procedimientos para acceder a ellas. Garantizan la exclusión mutua al permitir que solo un proceso esté activo dentro del monitor a la vez.

La elección de la solución adecuada para el problema de la sección crítica depende del entorno de programación, las características del hardware y los requisitos específicos de la aplicación. Es crucial comprender las ventajas, desventajas y posibles problemas (como el starvation) asociados con cada técnica para tomar una decisión informada.

# ¿Qué es una Referencia Crítica en Programación Concurrente?

En el contexto de la programación concurrente, una **referencia crítica** se refiere a la aparición de una variable dentro de una expresión o sentencia de asignación, donde dicha variable también es accedida (leída o escrita) por otro proceso o hilo concurrente. Es

decir, se trata de una variable compartida que puede ser modificada por más de un proceso o hilo al mismo tiempo.

### Importancia de las Referencias Críticas

La importancia de las referencias críticas radica en su potencial para introducir **interferencia** en un programa concurrente. La interferencia ocurre cuando las acciones de un proceso o hilo afectan el resultado de otro proceso o hilo que accede a las mismas variables compartidas.

En particular, las referencias críticas pueden llevar a **condiciones de carrera** (race conditions), donde el resultado de la ejecución del programa depende del orden impredecible en el que se ejecutan las instrucciones de los diferentes procesos o hilos.

### Manejo de Referencias Críticas

Para evitar la interferencia y las condiciones de carrera, es crucial identificar y manejar adecuadamente las referencias críticas en la programación concurrente.

La **propiedad de "A lo sumo una vez"** (ASV) [1, 2] es una herramienta útil para determinar si una expresión o sentencia de asignación es atómica, es decir, si se ejecuta como una única acción indivisible desde la perspectiva de otros procesos. Sin embargo, la propiedad ASV tiene limitaciones y no siempre es aplicable [3].

En casos más complejos, donde la propiedad ASV no es suficiente, se requieren **mecanismos de sincronización** adicionales, como semáforos o monitores, para controlar el acceso a las variables compartidas y garantizar la corrección del programa.

### Ejemplos de Referencias Críticas

El siguiente ejemplo, tomado de la fuente [4], ilustra el problema de las referencias críticas:

```
int x = 0, y = 0;  // Ninguna asignación satisface ASV
co
 x=y+1
//
 y=x+1
oc;
// Posibles resultados: x= 1 e y =2  /  x = 2 e y = 1
// Nunca debería ocurrir x = 1 e y = 1 → ERROR
```

En este ejemplo, ambas asignaciones (x=y+1 e y=x+1) contienen referencias críticas, ya que tanto x como y son variables compartidas que pueden ser modificadas por ambos procesos. Como consecuencia, el resultado final de la ejecución concurrente no está definido y puede llevar a inconsistencias.

### Resumen

En resumen, las referencias críticas son puntos críticos en la programación concurrente que requieren un manejo cuidadoso para evitar la interferencia y las condiciones de carrera. La propiedad ASV y los mecanismos de sincronización son herramientas esenciales para abordar estos desafíos y garantizar la corrección de los programas concurrentes.

# La Propiedad de "A lo sumo una vez" en la Programación Concurrente

La propiedad de "A lo sumo una vez" (ASV, por sus siglas en inglés) es un concepto crucial en la programación concurrente para garantizar la atomicidad de las operaciones,

especialmente en escenarios donde múltiples procesos o hilos acceden a datos compartidos. Esta propiedad establece que, para que una sección de código se considere atómica, debe cumplir con la siguiente restricción:
**"Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez dentro de la sección de código en cuestión".**

## Aplicación de la Propiedad ASV

Esta propiedad se aplica principalmente a las sentencias de asignación (x = e) y a las expresiones que no forman parte de una asignación.

**En una sentencia de asignación (x = e):**La propiedad ASV se cumple si la expresión (e) contiene a lo sumo una referencia a una variable que pueda ser modificada por otro proceso (referencia crítica), y la variable a la que se realiza la asignación (x) no es accedida por ningún otro proceso.
También se cumple si (e) no contiene referencias críticas, en cuyo caso (x) puede ser leída por otros procesos.
**En una expresión (e) fuera de una sentencia de asignación:** La propiedad ASV se cumple si (e) no contiene más de una referencia crítica.

## Importancia de la Propiedad ASV

La importancia de la propiedad ASV radica en su capacidad para asegurar que una sección de código se ejecute como si fuera una única acción indivisible (atómica) desde la perspectiva de otros procesos. Esto significa que:

**Consistencia de datos:** Se evita la aparición de resultados inesperados o erróneos debido a la interferencia entre procesos que acceden simultáneamente a la misma variable compartida.
**Simplificación del razonamiento sobre concurrencia:** Al garantizar que las operaciones se ejecutan atómicamente, se facilita la tarea de analizar y comprender el comportamiento de un programa concurrente, lo que a su vez simplifica la depuración y la verificación de su corrección.

## Ejemplos de la Propiedad ASV

Los siguientes ejemplos, extraídos de la fuente [1], ilustran la aplicación de la propiedad ASV:

**Ejemplo 1:**

```
int x=0, y=0; // No hay referencias críticas en ningún proceso
co
 x=x+1
//
 y=y+1
oc;
// En todas las historias x = 1 e y = 1
```

En este caso, ambas asignaciones (x=x+1 e y=y+1) cumplen la propiedad ASV porque no contienen referencias críticas. Por lo tanto, se garantiza que al finalizar la ejecución concurrente, x e y tendrán el valor 1.

**Ejemplo 2:**

```
int x = 0, y = 0; // El 1er proceso tiene 1 ref. crítica. El 2do ninguna.
co
 x=y+1
//
 y=y+1
```

```
oc;
// Siempre y = 1 y x = 1 o 2
```
Aquí, la asignación x=y+1 contiene una referencia crítica (y), pero la variable x no es accedida por el otro proceso. La asignación y=y+1 no contiene referencias críticas. En consecuencia, la propiedad ASV se cumple en ambos casos, asegurando que y siempre será 1, y x podrá ser 1 o 2, dependiendo del orden de ejecución.

**Ejemplo 3:**

```
int x = 0, y = 0;  // Ninguna asignación satisface ASV
co
 x=y+1
//
 y=x+1
oc;
// Posibles resultados: x= 1 e y =2  /  x = 2 e y = 1
// Nunca debería ocurrir x = 1 e y = 1 → ERROR
```
En este último ejemplo, ninguna de las asignaciones cumple con la propiedad ASV. Ambas expresiones (y+1 y x+1) contienen una referencia crítica, y las variables x e y son accedidas por ambos procesos. En consecuencia, el resultado de la ejecución concurrente no está definido y puede llevar a inconsistencias, como que x e y terminen con el valor 1.

### Limitaciones de la Propiedad ASV

Es importante destacar que la propiedad ASV no siempre es suficiente para garantizar la corrección de un programa concurrente. En situaciones donde se requiere ejecutar una secuencia de instrucciones como una única acción atómica, es necesario emplear mecanismos de sincronización adicionales, como semáforos o monitores, para garantizar la exclusión mutua y la sincronización por condición.

### Conclusión

La propiedad "A lo sumo una vez" es una herramienta valiosa en la programación concurrente para asegurar la atomicidad de operaciones simples y evitar inconsistencias en el acceso a datos compartidos. Sin embargo, su aplicación está limitada a casos específicos, y en escenarios más complejos, se requiere la utilización de mecanismos de sincronización más robustos para garantizar la corrección del programa.

# ¿Qué es la interferencia en programación concurrente y cómo se evita?

En la programación concurrente, la interferencia se produce cuando las acciones de un proceso invalidan las suposiciones hechas por otro proceso, lo que puede llevar a resultados incorrectos. Esto sucede porque los procesos concurrentes comparten recursos, como variables en memoria, y la modificación de estos recursos por un proceso puede afectar el comportamiento de otros.

La **interferencia** es un problema central en la programación concurrente porque puede ser difícil de predecir y depurar. A diferencia de los programas secuenciales, donde el orden de ejecución de las instrucciones está predeterminado, en la programación concurrente el orden de ejecución de los procesos puede variar, lo que aumenta las posibilidades de interferencia.

### Ejemplo de Interferencia

La fuente [Clase-2---Programacin-Concurrente---2024.pdf] proporciona un ejemplo claro de interferencia:

```
int X = 0
Process P1
{ int i
for [i=1 to N] → X=X+1
}
Process P2
{ int i
fa [i=1 to N] → X=X+1
}
```

En este ejemplo, dos procesos, P1 y P2, incrementan la variable compartida X N veces cada uno. En un escenario secuencial, el valor final de X sería 2N. Sin embargo, debido a la interferencia, el resultado final puede ser incorrecto.

La interferencia se produce porque la operación X=X+1 no es atómica, es decir, no se ejecuta como una única acción indivisible. En cambio, se compone de varias acciones más pequeñas:

**Cargar el valor de X en un registro.**

**Incrementar el valor del registro.**

**Almacenar el valor del registro en X.**

Si las acciones de P1 y P2 se intercalan durante la ejecución concurrente, es posible que un proceso sobrescriba el incremento realizado por el otro proceso, lo que lleva a un valor final de X menor que 2N.

## Cómo evitar la interferencia

Para evitar la interferencia y garantizar la corrección de los programas concurrentes, se utilizan diversas técnicas de sincronización. Algunas de las técnicas más comunes incluyen:

**Exclusión Mutua**: Esta técnica garantiza que solo un proceso pueda acceder a un recurso compartido a la vez. Se utilizan mecanismos como semáforos, monitores y locks (bloqueos) para implementar la exclusión mutua.

**Sincronización por Condición**: Esta técnica permite a los procesos esperar a que se cumpla una condición específica antes de continuar con su ejecución. Las variables de condición, junto con los monitores, se utilizan a menudo para implementar la sincronización por condición.

**Atomicidad de Grano Grueso**: Esta técnica implica combinar varias acciones en una sola acción atómica más grande. La idea es reducir la posibilidad de interferencia al minimizar el número de puntos en los que los procesos pueden intercalar sus acciones.

**Disjoint Variables**: Si es posible diseñar el programa de manera que los procesos concurrentes operen en conjuntos de variables completamente separados, se puede evitar la interferencia por completo. Sin embargo, esta técnica no siempre es aplicable.

## Conclusión

La interferencia es un desafío inherente a la programación concurrente. Comprender las causas de la interferencia y las técnicas para evitarla es esencial para desarrollar programas concurrentes correctos y eficientes.

Es importante recordar que las soluciones a la interferencia a menudo implican un compromiso entre la corrección, la eficiencia y la complejidad del programa. La elección de la técnica de sincronización adecuada dependerá de las características específicas del problema que se está resolviendo y de las restricciones del entorno de ejecución.

# Tipos de Barreras en la Programación Concurrente

Las barreras son mecanismos de sincronización utilizados en la programación concurrente para garantizar que un conjunto de procesos o hilos alcancen un punto específico en su ejecución antes de que ninguno de ellos pueda continuar. A continuación se explican tres tipos de barreras: Tree Barrier, Barrera Simétrica y Butterfly Barrier.

## Tree Barrier (Barrera de Árbol)

La **Tree Barrier** [1-3] organiza los procesos en una estructura de árbol jerárquica para la sincronización. Cada nodo del árbol representa un proceso, y la comunicación se produce entre nodos padre e hijo.

**Funcionamiento:**

Los procesos hoja del árbol envían señales de "arribo" a sus nodos padre.

Los nodos intermedios esperan a que todos sus hijos envíen la señal de "arribo" antes de propagar su propia señal al nodo padre.

Cuando el nodo raíz del árbol recibe señales de "arribo" de todos sus hijos, indica a sus hijos que pueden "continuar" con su ejecución.

La señal de "continuar" se propaga hacia abajo en el árbol, liberando a los procesos para que continúen su ejecución.

**Ventajas:**

Mayor eficiencia para un gran número de procesos en comparación con las barreras centralizadas, ya que el tiempo de propagación de las señales es proporcional a la altura del árbol (logarítmico con respecto al número de procesos).

Reducción de la contención en el nodo raíz en comparación con las barreras centralizadas.

**Desventajas:**

Los procesos en los nodos internos del árbol realizan más trabajo que los nodos hoja.

Requiere una estructura de comunicación más compleja que las barreras centralizadas.

## Barrera Simétrica

La **Barrera Simétrica** [4, 5] se caracteriza por que todos los procesos participantes realizan las mismas acciones durante la sincronización, lo que la hace especialmente adecuada para sistemas de memoria compartida con tiempos de acceso a memoria no uniformes.

**Funcionamiento:**

Se construye a partir de pares de barreras simples para dos procesos.

Cada par de procesos utiliza un mecanismo de espera mutua para sincronizarse.

Por ejemplo, cada proceso establece un flag al llegar a la barrera, espera a que el otro proceso establezca su flag y luego borra el flag del otro proceso [4].

Se requiere un esquema de interconexión para combinar las barreras de dos procesos y construir una barrera para *n* procesos.

**Ventajas:**

Mayor eficiencia en sistemas de memoria compartida con tiempos de acceso a memoria no uniformes.

Todos los procesos realizan las mismas acciones, lo que simplifica el código.

**Desventajas:**

Requiere un esquema de interconexión específico para combinar las barreras de dos procesos.
La complejidad del esquema de interconexión puede aumentar con el número de procesos.

### Butterfly Barrier (Barrera de Mariposa)

La **Butterfly Barrier** [3, 6-8] es un tipo específico de Barrera Simétrica que utiliza un patrón de interconexión similar al utilizado en la Transformada Rápida de Fourier.

**Funcionamiento:**
Se organiza en etapas logarítmicas con respecto al número de procesos.
En cada etapa, cada proceso se sincroniza con otro proceso a una distancia específica.
La distancia entre los procesos que se sincronizan se duplica en cada etapa.
Cuando todos los procesos han pasado por todas las etapas, se considera que han llegado a la barrera.
**Ventajas:**
Patrón de interconexión bien definido y eficiente.
Tiempo de sincronización logarítmico con respecto al número de procesos.
**Desventajas:**
El número de procesos debe ser una potencia de dos para un rendimiento óptimo.
Si no es así, se necesita agregar procesos "ficticios" o ajustar la lógica, lo que puede afectar la eficiencia [7].
Requiere un esquema de gestión de concurrencia para evitar condiciones de carrera en el acceso a las banderas de sincronización [9].

## Conclusión

La elección del tipo de barrera más adecuado depende de las características específicas del sistema y la aplicación. Las Tree Barriers son eficientes para un gran número de procesos, mientras que las Barreras Simétricas, como la Butterfly Barrier, son más apropiadas para sistemas de memoria compartida con tiempos de acceso a memoria no uniformes. La Butterfly Barrier, con su patrón de interconexión específico, ofrece un buen rendimiento cuando el número de procesos es una potencia de dos.

## Características para la aparición de un Deadlock

Un deadlock, también conocido como interbloqueo o abrazo mortal, se produce en sistemas concurrentes cuando dos o más procesos quedan bloqueados indefinidamente, esperando a que el otro libere los recursos que necesita para continuar. Para que ocurra un deadlock, se deben cumplir cuatro condiciones necesarias, tal como se describe en la fuente [1, 2]:

**1. Recursos reusables serialmente:** Los procesos involucrados deben compartir recursos que solo se puedan utilizar en exclusión mutua. Esto significa que un recurso solo puede ser utilizado por un proceso a la vez.

**2. Adquisición incremental:** Los procesos deben mantener los recursos que ya poseen mientras esperan adquirir recursos adicionales. Esto crea una situación en la que un proceso puede bloquearse esperando un recurso que está siendo retenido por otro proceso que, a su vez, está esperando un recurso retenido por el primero.

**3. No-preemption:** Los recursos, una vez adquiridos por un proceso, no pueden ser arrebatados de manera forzada. Solo pueden ser liberados voluntariamente por el proceso que los posee.

**4. Espera cíclica:** Debe existir una cadena circular de procesos, donde cada proceso está esperando un recurso que está siendo retenido por el siguiente proceso en la cadena, y el último proceso de la cadena está esperando un recurso retenido por el primero.
**Es importante destacar que la presencia de estas cuatro condiciones simultáneamente es necesaria y suficiente para que ocurra un deadlock**. Si alguna de estas condiciones no se cumple, el deadlock no puede ocurrir. Por lo tanto, las estrategias para prevenir deadlocks se basan en romper una o más de estas condiciones.

# La Granularidad en la Informática

La **granularidad**, en el contexto de la computación paralela y distribuida, se refiere al **tamaño relativo de las unidades de trabajo o tareas** en las que se divide un problema para su procesamiento. [1-3]
A continuación, se detallan los dos tipos principales de granularidad:

- **Granularidad fina (Fine-grained):** Implica descomponer el problema en un gran número de tareas pequeñas. [1] Este enfoque generalmente conduce a un mayor grado de concurrencia, lo que significa que se pueden ejecutar más tareas simultáneamente. [2, 4] Un ejemplo típico es la multiplicación matriz-vector, donde cada tarea podría ser el cálculo de un único producto punto. [1]
- **Granularidad gruesa (Coarse-grained):** Se refiere a dividir el problema en un número menor de tareas más grandes. [1] Este enfoque generalmente resulta en un menor grado de concurrencia. [2] Siguiendo con el ejemplo de la multiplicación matriz-vector, una granularidad gruesa podría implicar que cada tarea calcule un bloque completo de la matriz resultante. [1]

## Factores a considerar al determinar la granularidad:

La elección de la granularidad adecuada depende de varios factores, y a menudo implica un compromiso entre diferentes aspectos del rendimiento:

- **Comunicación e interacción:** Las tareas de granularidad fina suelen implicar una mayor comunicación e interacción entre sí, ya que es más probable que compartan datos o necesiten sincronizarse. [5, 6] En contraste, las tareas de granularidad gruesa tienden a ser más independientes y, por lo tanto, requieren menos comunicación. [5]
- **Sobrecarga:** Un gran número de tareas de granularidad fina puede aumentar la sobrecarga del sistema, ya que se necesita tiempo para administrar y coordinar todas las tareas. [6, 7]
- **Balanceo de carga:** Una granularidad más fina puede facilitar el balanceo de carga, ya que las tareas más pequeñas se pueden distribuir de manera más uniforme entre los procesadores. [8]
- **Localidad de datos:** Las tareas de granularidad gruesa pueden aprovechar mejor la localidad de datos, ya que es más probable que accedan a los mismos datos de forma consecutiva. [8]

## Resumen

En resumen, la granularidad es un factor crucial en el diseño de algoritmos paralelos y distribuidos. Elegir la granularidad adecuada implica equilibrar el grado de concurrencia, la sobrecarga de comunicación, el balanceo de carga y la localidad de datos para lograr un rendimiento óptimo.

# Explicación de "Passing the Baton"

"Passing the Baton" es una técnica de programación concurrente utilizada para implementar sentencias await complejas y controlar el orden en que los procesos se despiertan después de estar bloqueados. Esta técnica emplea semáforos binarios divididos (SBS) para proporcionar exclusión mutua y señalizar a los procesos en espera.

**El concepto central detrás de "Passing the Baton" es la idea de un "testigo" (baton) que se pasa entre los procesos**. Cuando un proceso está dentro de una sección crítica (SC), se considera que posee el testigo. Al salir de la SC, el proceso evalúa las condiciones de espera de los demás procesos y "pasa el testigo" a uno de los procesos que está esperando por una condición que ahora se cumple.

**El mecanismo de "Passing the Baton" se implementa de la siguiente manera:**

**Semáforo de entrada**: Se utiliza un semáforo binario principal (e) para proteger el acceso a la sección crítica y a la lógica de "Passing the Baton".

**Semáforos de condición**: Se crean semáforos binarios adicionales (b1, b2, ..., bn) para cada condición de espera.

**Contadores de espera**: Se asocia un contador (d1, d2, ..., dn) a cada semáforo de condición, que indica cuántos procesos están esperando por esa condición.

**Protocolo de entrada**: Un proceso que desea entrar a la SC ejecuta una operación P(e) para adquirir el semáforo de entrada.

**Evaluación de condiciones**: El proceso evalúa las condiciones de espera (B1, B2, ..., Bn) asociadas a los semáforos de condición. Si la condición Bi no se cumple, el proceso incrementa el contador de espera di, libera el semáforo de entrada (V(e)) y se bloquea en el semáforo de condición bi (P(bi)).

**Protocolo de salida (SIGNAL)**: Un proceso que sale de la SC ejecuta una sección de código llamada SIGNAL.

**Passing the Baton**: SIGNAL evalúa las condiciones de espera y, si alguna condición Bi se cumple y el contador de espera di es mayor que 0, realiza una operación V(bi) para despertar a un proceso en espera. Si ninguna condición se cumple o no hay procesos esperando, SIGNAL libera el semáforo de entrada (V(e)).

**"Passing the Baton" permite implementar await arbitrarios y controlar el orden en que los procesos se despiertan, lo que es crucial para evitar la inanición y garantizar la equidad.**

**Ejemplo de "Passing the Baton" en el problema de lectores y escritores (fuentes [1-9]):**

**Lectores**: pueden acceder al recurso simultáneamente si no hay escritores activos.

**Escritores**: necesitan acceso exclusivo al recurso.

Se utilizan los siguientes semáforos y contadores:

e: semáforo de entrada.
r: semáforo para bloquear lectores.
w: semáforo para bloquear escritores.
nr: número de lectores activos.
nw: número de escritores activos.
dr: número de lectores esperando.
dw: número de escritores esperando.

**Protocolo de entrada del lector:**

P(e): adquirir el semáforo de entrada.
Si nw > 0 o dw > 0: dr = dr + 1; V(e); P(r).
nr = nr + 1.
SIGNAL.
Acceder al recurso.

**Protocolo de entrada del escritor:**

P(e): adquirir el semáforo de entrada.
Si nr > 0 o nw > 0: dw = dw + 1; V(e); P(w).
nw = nw + 1.
SIGNAL.
Acceder al recurso.

**Protocolo de salida (SIGNAL):**

P(e): adquirir el semáforo de entrada.
Si nw == 0 y dr > 0: dr = dr - 1; V(r).
Si nr == 0 y nw == 0 y dw > 0: dw = dw - 1; V(w).
Si no se cumple ninguna de las condiciones anteriores: V(e).

**En este ejemplo, "Passing the Baton" se utiliza para garantizar que los escritores no sean bloqueados indefinidamente por los lectores.** Cuando un lector sale de la SC, el código SIGNAL comprueba si hay algún escritor esperando. Si es así, despierta al escritor; de lo contrario, despierta a otro lector si lo hay.
La técnica "Passing the Baton" también se utiliza para implementar otras políticas de sincronización, como la **asignación de recursos** y la **sincronización por condición**.
En resumen, "Passing the Baton" es una técnica poderosa para implementar la sincronización compleja en la programación concurrente.

# Entendiendo "Passing the Condition" en la Programación Concurrente

"Passing the condition" es una técnica utilizada en la programación concurrente para gestionar la sincronización entre procesos que utilizan monitores. La idea central es que, en lugar de forzar a un proceso a verificar continuamente una condición, un proceso "pasa" la responsabilidad de señalar cuando esa condición se cumple a otro proceso.
Si bien el término "passing the condition" no se menciona explícitamente en las fuentes, el concepto se describe en el contexto de la simulación de semáforos utilizando monitores [1, 2]. Las fuentes explican cómo un monitor puede simular el comportamiento de un semáforo FIFO utilizando una variable de condición.

**Cómo funciona:**

**Monitores:** Los monitores son estructuras de datos que encapsulan recursos compartidos y proporcionan métodos sincronizados para acceder a esos recursos [3].

**Variables de condición:** Dentro de un monitor, las variables de condición se utilizan para bloquear procesos hasta que se cumpla una determinada condición [4, 5].

**Passing the condition:** Un proceso que modifica el estado del recurso compartido y, por lo tanto, potencialmente satisface la condición de espera de otro proceso, utiliza una operación signal en la variable de condición apropiada. Esta señal "pasa" la condición al proceso en espera, que luego puede reanudar su ejecución.

**Ejemplo: Semáforo FIFO**

En el ejemplo del monitor FIFOsemaphore [2], la variable de condición pos se utiliza para señalizar cuando el valor del semáforo s es mayor que 0. La operación Vsem incrementa el valor de s y, si hay algún proceso esperando en pos, realiza una operación signal(pos). Esto "pasa" la condición "s > 0" al proceso en espera, permitiéndole continuar.

**Beneficios de "Passing the Condition":**

**Mayor eficiencia:** Evita el "busy waiting", donde los procesos verifican repetidamente una condición que aún no se cumple.
**Mejor legibilidad:** Simplifica el código al encapsular la lógica de señalización dentro del monitor.
**Menor riesgo de errores:** Reduce la posibilidad de que los procesos se pierdan señales o creen condiciones de carrera.

**Diferencias con "Passing the Baton":**

Si bien ambas técnicas se utilizan para la sincronización, hay diferencias clave:

**"Passing the baton"** generalmente se refiere a la técnica de pasar un "testigo" entre procesos que utilizan semáforos para controlar el acceso a una sección crítica.
**"Passing the condition"** se utiliza específicamente en monitores con variables de condición para señalar cuando se cumple una condición específica.

**En resumen, "passing the condition" es una técnica poderosa para la sincronización en la programación concurrente que mejora la eficiencia, la legibilidad y la corrección del código.**

# Ejemplo de Instrucción Fetch-and-Add (FA) y su Explicación

La instrucción **Fetch-and-Add (FA)** es una operación atómica que se utiliza frecuentemente en la programación concurrente para la sincronización entre procesos, especialmente en escenarios donde se necesita un acceso seguro a contadores compartidos.

**Ejemplo de FA en un Algoritmo de Ticket:**

El algoritmo de ticket es un método común para implementar la exclusión mutua. En este algoritmo, cada proceso que desea entrar en la sección crítica recibe un número de ticket único. El proceso con el ticket más bajo puede entrar en la sección crítica.

El código siguiente muestra un ejemplo de cómo se puede usar FA para implementar el algoritmo de ticket:

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );
process SC [i: 1..n] {
 while (true) {
   turno[i] = FA (numero, 1);
   while (turno[i] <> proximo) skip;
   // Sección crítica
   proximo = proximo + 1;
   // Sección no crítica
 }
}
```

**Explicación:**

**Inicialización:**
numero: Un contador global que se utiliza para generar tickets. Se inicializa en 1.
proximo: Indica el número de ticket del proceso que puede entrar en la sección crítica. Se inicializa en 1.
turno[1:n]: Un arreglo que almacena los números de ticket de los procesos. Se inicializa con todos los elementos en 0.
**Obtener un Ticket (turno[i] = FA(numero, 1);):**

Cuando un proceso quiere entrar en la sección crítica, ejecuta la instrucción FA(numero, 1).
Esta instrucción realiza dos acciones atómicamente:
Incrementa el valor de numero en 1.
Devuelve el valor **original** de numero **antes** del incremento y lo almacena en la variable turno[i]. De este modo, cada proceso obtiene un ticket único.
**Esperar el Turno (while (turno[i] <> proximo) skip;):**
El proceso espera en un bucle (busy-waiting) mientras que su número de ticket (turno[i]) no sea igual al valor de proximo.
Esto garantiza que los procesos entren en la sección crítica en el orden en que obtuvieron sus tickets.
**Entrar en la Sección Crítica:**
Cuando turno[i] es igual a proximo, el proceso puede entrar en la sección crítica.
**Salir de la Sección Crítica y Avanzar el Turno (proximo = proximo + 1;):**
Al salir de la sección crítica, el proceso incrementa proximo en 1, permitiendo que el siguiente proceso en la cola pueda entrar.

**Ventajas de FA:**

**Atomicidad:** La operación FA se ejecuta atómicamente, lo que significa que no puede ser interrumpida por otro proceso. Esto es fundamental para evitar condiciones de carrera.
**Eficiencia:** FA se suele implementar directamente en hardware, lo que la convierte en una operación muy eficiente.

**En resumen**, la instrucción Fetch-and-Add es una herramienta valiosa en la programación concurrente. Su atomicidad y eficiencia la hacen ideal para la implementación de algoritmos como el de tickets, que permiten una gestión ordenada y segura del acceso a recursos compartidos. [1-3]

# Semántica de AWAIT y su Relación con TEST & SET y FETCH & ADD

La instrucción de grano grueso **AWAIT** se utiliza en programación concurrente para especificar la sincronización entre procesos, permitiendo que un proceso se demore hasta que una condición específica se cumpla. Su sintaxis general es:
await (B) S;
Donde:

**B** es una expresión booleana que define la condición de espera.
**S** es una secuencia de sentencias que se ejecutarán atómicamente una vez que la condición **B** sea verdadera.

**Semántica:**
La instrucción AWAIT garantiza que la expresión **B** será verdadera antes de que se ejecute la secuencia de sentencias **S**. Mientras la condición **B** sea falsa, el proceso que ejecuta AWAIT se bloqueará, liberando el acceso al monitor o a la sección crítica para que otros procesos puedan avanzar. Una vez que la condición **B** se vuelve verdadera, el proceso en espera se reanuda y ejecuta la secuencia **S** de forma atómica, evitando la interferencia de otros procesos.
**Relación con TEST & SET y FETCH & ADD:**

Las instrucciones de hardware como **TEST & SET (TS)** y **FETCH & ADD (FA)** son fundamentales para la implementación eficiente de AWAIT, especialmente en la creación de mecanismos de exclusión mutua como **spin locks**.

**TEST & SET:**

TS es una instrucción atómica que prueba y modifica el valor de una variable booleana.

Se utiliza para implementar **spin locks**, donde un proceso "gira" en un bucle mientras que el valor de una variable de bloqueo (lock) es verdadero, indicando que la sección crítica está ocupada.

El proceso que intenta acceder a la sección crítica utiliza TS para verificar y, si es posible, cambiar el valor del bloqueo a verdadero. Si el valor ya era verdadero, el proceso sigue girando en el bucle.

**FETCH & ADD:**

FA es una instrucción atómica que incrementa el valor de una variable y devuelve el valor original.

Se utiliza en la implementación de mecanismos de sincronización como barreras y algoritmos de tickets.

En una barrera, por ejemplo, cada proceso realiza un FA sobre un contador compartido. El proceso que realiza el último incremento libera a los demás procesos que están esperando en la barrera.

**Implementación de AWAIT:**

La instrucción AWAIT se puede implementar utilizando TS o FA para crear una sección crítica alrededor de la comprobación de la condición **B** y la ejecución de la secuencia **S**. La idea es utilizar una variable de bloqueo que se modifica atómicamente con TS o FA.

**Ejemplo con TEST & SET:**

```
bool lock = false;
...
<await (not lock) S;>
// Implementación con TS:
while (TS(lock)) skip;  // Spin lock
S;
lock = false;
```

**Ejemplo con FETCH & ADD (Algoritmo de Ticket):**

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );
...
< await turno[i] == proximo; >
// Implementación con FA:
turno[i] = FA (numero, 1);  // Obtener un ticket
while (turno[i] <> proximo) skip;  // Esperar el turno
```

En resumen, la instrucción de grano grueso **AWAIT** proporciona una forma abstracta y poderosa de expresar la sincronización en programas concurrentes. Su implementación eficiente se basa en instrucciones atómicas de hardware como **TEST & SET** y **FETCH & ADD** para crear secciones críticas y garantizar la exclusión mutua en el acceso a los recursos compartidos.

# Definición de Concurrencia y Diferencias con el Procesamiento Secuencial y Paralelo

La **concurrencia** es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente, permitiendo que distintos objetos actúen al mismo tiempo [1]. Este concepto es relevante para el diseño de hardware, sistemas operativos, multiprocesadores, computación distribuida, programación y diseño [2].
A continuación, se diferencian los tres tipos de procesamiento:

**Procesamiento secuencial:** En este modelo, solo hay un flujo de control, ejecutando una instrucción a la vez, una tras otra [3]. Es similar a un programa monoprocesador, donde las instrucciones se ejecutan en un orden estricto [4]. Un ejemplo sería la fabricación de un objeto en una sola máquina: cada parte debe ser fabricada una tras otra, en un orden temporal fijo [5].

**Procesamiento concurrente:** Este modelo involucra múltiples hilos o flujos de control que se ejecutan "al mismo tiempo" [4]. Sin embargo, esto no implica necesariamente que se ejecuten en paralelo en distintos procesadores. Un programa concurrente puede ejecutarse en un único procesador, donde el sistema operativo se encarga de compartir el tiempo de CPU entre los distintos procesos [6]. En el ejemplo de la fabricación, sería como si una sola máquina dedicara una parte de su tiempo a cada componente del objeto [7].

**Procesamiento paralelo:** Este modelo se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución [8]. Volviendo al ejemplo de la fabricación, sería como tener una máquina dedicada a la fabricación de cada parte del objeto, trabajando simultáneamente [9]. Esto permite completar el trabajo en menos tiempo y con menor esfuerzo individual [9].

En resumen, la **concurrencia** es un concepto de software que no depende del hardware [6, 8], mientras que el **paralelismo** implica la ejecución en múltiples procesadores [8]. Un programa **concurrente** puede o no ejecutarse en **paralelo**, dependiendo de la disponibilidad de múltiples procesadores.

# Tipos de Fairness en Programación Concurrente

La **fairness** en programación concurrente busca garantizar que todos los procesos tengan la oportunidad de progresar sin importar el comportamiento de los demás. Las fuentes describen tres tipos principales de fairness:

**Fairness Incondicional:** Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que está lista para ejecutarse (elegible) eventualmente se ejecuta. Esto significa que, si una acción no tiene condiciones que impidan su ejecución, eventualmente el sistema le dará la oportunidad de ejecutarse. Un ejemplo de esto es la política Round Robin en un monoprocesador, donde cada proceso tiene su turno para ejecutarse [1].

**Fairness Débil:** Esta política va un paso más allá de la fairness incondicional. Además de garantizar la ejecución de acciones incondicionales, también asegura que una acción atómica condicional se ejecute eventualmente si su condición se vuelve verdadera y permanece así hasta que el proceso pueda verla. Sin embargo, esta política no puede garantizar que la acción se ejecute si la condición cambia de valor (de falso a verdadero y de nuevo a falso) mientras el proceso está esperando [2].

**Fairness Fuerte:** Es el tipo de fairness más robusto. Al igual que las anteriores, garantiza la ejecución de acciones incondicionales. Para las acciones condicionales, asegura su ejecución si la condición se vuelve verdadera con una

frecuencia infinita. En otras palabras, incluso si la condición cambia de valor, si se vuelve verdadera infinitas veces, el proceso eventualmente podrá ejecutar la acción [3].

En resumen, la **fairness** juega un papel fundamental en el diseño de sistemas concurrentes al asegurar que ningún proceso quede en espera indefinidamente. Cada tipo de fairness ofrece un nivel diferente de garantía, siendo la **fairness fuerte** la más robusta, pero también la más compleja de implementar en la práctica.