Teoría, conceptos importantes

Propiedades de Seguridad en la Programación Concurrente

Una propiedad de seguridad en la programación concurrente se refiere a un atributo que se mantiene verdadero en todos los posibles estados de la ejecución de un programa. En esencia, afirma que nada "malo" sucederá durante la ejecución del programa.

Las propiedades de seguridad en la programación concurrente aseguran que el programa siempre esté en un estado consistente y evitan resultados no deseados debido a la interacción entre procesos concurrentes. Si se viola una propiedad de seguridad, hay un error en el diseño o la implementación. Ejemplos:

1. **Exclusión Mutua**: Asegura que solo un proceso acceda a un recurso compartido a la vez, evitando accesos simultáneos.
2. **Ausencia de Interferencia**: Limita la interacción destructiva entre procesos para prevenir resultados incorrectos.
3. **Ausencia de Deadlock**: Evita situaciones donde procesos se bloquean indefinidamente esperando recursos.
4. **Corrección Parcial**: Garantiza que, si el programa termina, el estado final será correcto, aunque no garantiza que siempre termine.

Las herramientas para garantizar las propiedades de seguridad en la programación concurrente incluyen:

1. **Semáforos**: Variables especiales que controlan el acceso a recursos compartidos y sincronizan procesos mediante operaciones de espera y señalización, evitando condiciones de carrera.
2. **Monitores**: Estructuras que encapsulan datos compartidos y proporcionan procedimientos sincronizados para garantizar acceso exclusivo a un proceso a la vez.
3. **Locks**: Mecanismos de sincronización que permiten a un solo proceso adquirir acceso exclusivo a un recurso mientras lo necesite.
4. **Barreras**: Puntos de sincronización que aseguran que todos los procesos deben llegar a ellos antes de que cualquiera continúe, facilitando una ejecución coordinada.

Explicación de la Propiedad de Vida con Ejemplos

En la programación concurrente, una propiedad de vida o vivacidad asegura que el programa seguirá progresando y que eventos deseados ocurrirán. A diferencia de las propiedades de seguridad, que evitan eventos "malos", las propiedades de vida aseguran el progreso continuo del programa. Ejemplos:

1. **Terminación**: Asegura que el programa o proceso terminará eventualmente, evitando bucles infinitos.

2. **Acceso Eventual a un Recurso**: Garantiza que un proceso que solicita un recurso lo obtendrá eventualmente.
3. **Entrega de Mensajes**: En sistemas de paso de mensajes, asegura que un mensaje enviado llegará a su destino.
4. **Ausencia de Starvation**: Previene que un proceso se quede esperando indefinidamente mientras otros progresan.
5. **Fairness**: Asegura que todos los procesos tienen una oportunidad justa de avanzar. Variantes incluyen:
    - *Fairness Incondicional*: Cada proceso tiene infinitas oportunidades de ejecutar su acción.
    - *Fairness Débil*: Una acción habilitada se ejecutará eventualmente.
    - *Fairness Fuerte*: Una acción habilitada infinitamente se ejecutará eventualmente.

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

SC (Signal and Continue) permite que el proceso señalizador continúe ejecutándose, mientras que el proceso que estaba en espera debe competir por el monitor.
SW (Signal and Wait) cede el control del monitor de inmediato al proceso en espera, asegurando que este continúe su ejecución inmediatamente, aunque añade algo de complejidad en términos de implementación.

El problema de la sección crítica en programación concurrente

Problema de la Sección Crítica

Este problema es central en la programación concurrente, especialmente cuando múltiples procesos o hilos comparten recursos. Se busca evitar que dos procesos accedan simultáneamente a una sección crítica del código, lo que podría generar condiciones de carrera y resultados impredecibles. El objetivo es diseñar protocolos que garanticen:

1. **Exclusión Mutua**: Solo un proceso puede estar en la sección crítica al mismo tiempo.
2. **Ausencia de Deadlock**: Si varios procesos intentan entrar, al menos uno debe tener éxito.
3. **Ausencia de Demora Innecesaria**: Un proceso no debe ser bloqueado si los demás están fuera de la sección crítica.
4. **Entrada Eventual**: Un proceso que quiera acceder a la sección crítica debe poder hacerlo eventualmente.

Soluciones al Problema

Existen diversas soluciones con diferentes enfoques, cada una con sus pros y contras:

- **Deshabilitar Interrupciones**: Adecuado para sistemas de un solo procesador, pero no viable en sistemas multiprocesador.
- **Bloqueo de Grano Grueso**: Usa variables de bloqueo compartidas; efectivo entre dos procesos, pero no siempre garantiza exclusión mutua en casos complejos.
- **Spin Locks**: Utilizan instrucciones atómicas como Test-and-Set para controlar el acceso, pero pueden aumentar la contención de memoria.
- **Test-and-Test-and-Set (TTS)**: Mejora el Test-and-Set reduciendo la contención.

Algoritmos Justos (tie-breaker, ticket, bakery):

1. Algoritmo Tie-Breaker (Algoritmo de Peterson)

- **Descripción**: Diseñado para dos procesos, utiliza una variable "ultimo" para rastrear cuál proceso ingresó a su sección crítica más recientemente. Esto previene bloqueos indefinidos al dar prioridad al proceso que no fue el último.
- **Implementación**: Incluye variables booleanas "in1" e "in2" para indicar la intención de ingresar y un bucle que espera condiciones para entrar en la sección crítica.
- **Generalización a n procesos**: Se puede extender mediante etapas múltiples, donde cada etapa utiliza el algoritmo para decidir qué procesos avanzan, asegurando que solo un proceso avance por etapa.

2. Algoritmo Ticket

- **Descripción**: Funciona como un sistema de tickets; cada proceso obtiene un número único y el de menor valor tiene prioridad para entrar en la sección crítica.
- **Implementación**: Usa un contador global "numero" para generar tickets y una variable "proximo" para identificar el ticket del proceso que puede ingresar. Un arreglo "turno" almacena los tickets.
- **Requisito de la instrucción Fetch-and-Add**: La implementación eficiente necesita una instrucción atómica para actualizar el contador, aunque sin ella la equidad no está garantizada.

3. Algoritmo Bakery

- **Descripción**: Variación del algoritmo Ticket que no necesita instrucciones especiales. Los procesos se autoasignan un número mayor al máximo actual entre ellos.
- **Implementación**: Similar al Ticket, usa un arreglo "turno" para los números de ticket, donde cada proceso calcula el máximo actual y lo incrementa.
- **Complejidad y equidad**: Más complejo que el Ticket, garantiza equidad sin instrucciones especiales, pero puede enfrentar problemas de desbordamiento de enteros si se ejecuta por mucho tiempo.

Alternativas a los Spin Locks

Otros mecanismos como semáforos y monitores ofrecen soluciones más estructuradas:

- **Semáforos**: Variables enteras con operaciones atómicas para controlar el acceso, permitiendo sincronización entre procesos.
- **Monitores**: Estructuras de datos que encapsulan el acceso a variables compartidas, asegurando exclusión mutua dentro del monitor.

¿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.

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

Propiedad "A lo sumo una vez" (ASV)

La propiedad ASV asegura la atomicidad en operaciones concurrentes, permitiendo que las secciones de código actúen como una única acción indivisible cuando acceden a datos compartidos. Esta propiedad indica que, para que el código sea considerado atómico, debe cumplir con:

- **Restricción de Referencia Única**: Solo puede haber una variable compartida y puede ser referenciada una sola vez en la sección de código.

Aplicación de la Propiedad ASV

ASV se aplica a sentencias de asignación y expresiones:

1. **En Asignaciones (x = e)**:
    - Se cumple si **(e)** contiene como máximo una referencia crítica y la variable **(x)** no es accedida por otros procesos.
    - También se cumple si **(e)** no contiene referencias críticas y **(x)** es leída por otros procesos.
2. **En Expresiones fuera de Asignaciones**:
    - Se cumple si la expresión tiene como máximo una referencia crítica.

Importancia de ASV

La propiedad ASV ayuda a:

- **Mantener Consistencia de Datos**: Evita resultados inesperados debidos a la interferencia entre procesos.
- **Simplificar el Análisis Concurrente**: Facilita el razonamiento sobre el comportamiento del programa concurrente.

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

La interferencia en la programación concurrente ocurre cuando las acciones de un proceso afectan negativamente el comportamiento de otro, al invalidar sus suposiciones sobre el estado de los recursos compartidos. Este problema surge porque varios procesos pueden modificar los mismos recursos, como variables en memoria, lo que puede llevar a resultados inesperados.

Por ejemplo, si dos procesos incrementan una variable compartida, el resultado final puede ser incorrecto debido a que la operación de incremento no es atómica. Esto se debe a que consta de varias etapas: cargar, incrementar y almacenar el valor. Si las acciones de los procesos se intercalan, un proceso puede sobrescribir el incremento del otro, resultando en un valor menor al esperado.

Para evitar la interferencia, se emplean varias técnicas de sincronización:

1. **Exclusión Mutua**: Asegura que solo un proceso pueda acceder a un recurso compartido a la vez, usando mecanismos como semáforos, monitores y bloqueos (locks).
2. **Sincronización por Condición**: Permite que los procesos esperen a que se cumplan ciertas condiciones antes de continuar, frecuentemente implementada con variables de condición y monitores.
3. **Atomicidad de Grano Grueso**: Combina varias acciones en una sola operación atómica para reducir puntos de intercalado.
4. **Disjoint Variables**: Si se pueden separar las variables compartidas entre procesos, se puede evitar la interferencia por completo.

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.

- **Barrera con contador compartido:** Este es el tipo más simple de barrera. Se utiliza un contador compartido para rastrear cuántos procesos han llegado a la barrera. Cada proceso incrementa el contador cuando llega a la barrera y espera hasta que el contador alcanza el número total de procesos. Una vez que el contador llega al número total de procesos, todos los procesos son liberados.

- **Barrera con flags y coordinador:** En esta implementación, se utiliza un arreglo de flags, uno para cada proceso, para indicar si un proceso ha llegado a la barrera. Un proceso coordinador independiente se encarga de comprobar si todos los procesos han establecido su flag. Cuando todos los flags están establecidos, el coordinador libera a todos los procesos.
- **Barrera tipo árbol:** En esta implementación, los procesos se organizan en una estructura de árbol. Cada nodo del árbol representa un proceso y cada proceso se sincroniza con sus procesos hijos y padre. La raíz del árbol libera a todos los procesos una vez que todos los procesos han llegado a la barrera.
- **Barrera simétrica (Butterfly Barrier):** Esta barrera es específica para sistemas con un número de procesos igual a una potencia de dos. En este tipo de barrera, los procesos se sincronizan en pares en cada etapa. En cada etapa, la distancia entre los procesos que se sincronizan se duplica hasta que todos los procesos se han sincronizado entre sí.

Características para la aparición de un Deadlock

Un **deadlock** o **interbloqueo** ocurre en sistemas concurrentes cuando dos o más procesos quedan bloqueados indefinidamente, cada uno esperando a que el otro libere recursos necesarios para continuar. Para que se produzca un deadlock, deben cumplirse las siguientes cuatro condiciones de manera simultánea:

1. **Recursos reusables serialmente**: Los procesos deben compartir recursos que solo pueden ser utilizados en **exclusión mutua**, es decir, un recurso solo puede estar disponible para un proceso a la vez.
2. **Adquisición incremental**: Los procesos retienen los recursos ya obtenidos mientras esperan adquirir recursos adicionales. Esto puede generar una cadena de bloqueos en la que un proceso A espera por un recurso que está siendo retenido por un proceso B, el cual, a su vez, espera un recurso retenido por el proceso A.
3. **No-preemption**: Los recursos no pueden ser retirados forzadamente de un proceso que los ha adquirido. Solo se liberan cuando el proceso decide hacerlo voluntariamente.
4. **Espera cíclica**: Debe existir una cadena circular de procesos, donde cada proceso espera un recurso retenido por el siguiente proceso en la cadena, con el último proceso esperando un recurso que posee el primer proceso.

La presencia de estas condiciones es **necesaria y suficiente** para que ocurra un deadlock. Por lo tanto, para prevenir deadlocks, es esencial romper al menos una de estas condiciones.

La Granularidad en la Informática

En el contexto de la computación paralela y distribuida, **la granularidad** se refiere al tamaño relativo de las unidades de trabajo o tareas en las que se divide un problema para su procesamiento. Existen dos tipos principales de granularidad:

1. **Granularidad fina (Fine-grained)**:
    - Implica dividir el problema en un gran número de tareas pequeñas.

    - Permite un alto grado de concurrencia, ya que se pueden ejecutar más tareas simultáneamente.
    - Ejemplo: En una multiplicación matriz-vector, cada tarea puede ser el cálculo de un único producto punto.
2. **Granularidad gruesa (Coarse-grained)**:
    - Se refiere a dividir el problema en un número menor de tareas más grandes.
    - Esto reduce el grado de concurrencia en comparación con la granularidad fina.
    - Ejemplo: En la multiplicación matriz-vector, una tarea podría ser el cálculo de un bloque completo de la matriz resultante.

Factores a considerar al determinar la granularidad

La elección de la granularidad adecuada depende de varios factores y debe equilibrar diferentes aspectos del rendimiento:

- **Comunicación e interacción**: Las tareas de granularidad fina suelen requerir más comunicación e interacción, mientras que las tareas de granularidad gruesa son más independientes.
- **Sobrecarga**: Un gran número de tareas pequeñas puede aumentar la sobrecarga del sistema debido al tiempo necesario para gestionar y coordinar dichas tareas.
- **Balanceo de carga**: Las tareas más pequeñas permiten un mejor balanceo de carga, ya que pueden distribuirse uniformemente entre los procesadores.
- **Localidad de datos**: Las tareas grandes pueden aprovechar mejor la localidad de datos, ya que tienden a acceder a datos consecutivos.

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.

Passing the Condition

""Passing the Baton" es una técnica de programación concurrente que se utiliza para gestionar sentencias await complejas y controlar el orden de activación de procesos bloqueados. Utiliza semáforos binarios divididos (SBS) para garantizar la exclusión mutua y facilitar la señalización a los procesos en espera.

Conceptos Clave:

- **Testigo (Baton):** Representa el control de acceso a la sección crítica (SC) que se pasa entre procesos. Cuando un proceso entra en la SC, se dice que tiene el testigo. Al salir, evalúa las condiciones de otros procesos y pasa el testigo a uno que esté esperando por una condición cumplida.

Implementación:

1. **Semáforo de Entrada (e):** Semáforo principal para proteger el acceso a la SC y la lógica de "Passing the Baton".
2. **Semáforos de Condición (b1, b2, ..., bn):** Cada uno corresponde a una condición de espera.
3. **Contadores de Espera (d1, d2, ..., dn):** Indican cuántos procesos esperan por cada condición.

Protocolo de Entrada:

- Un proceso que desea entrar a la SC ejecuta P(e) para adquirir el semáforo de entrada.
- Evalúa las condiciones. Si no se cumple, incrementa el contador de espera, libera el semáforo de entrada (V(e)), y se bloquea en el semáforo de condición correspondiente (P(bi)).

Protocolo de Salida (SIGNAL):

- Al salir de la SC, el proceso ejecuta una sección de código llamada SIGNAL.
- Este evalúa las condiciones de espera y, si alguna se cumple y hay procesos esperando, despierta uno de ellos. Si no, libera el semáforo de entrada.

Ejemplo: Problema de Lectores y Escritores

- **Lectores:** Pueden acceder al recurso simultáneamente si no hay escritores.
- **Escritores:** Necesitan acceso exclusivo.

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.

Protocolos:

- **Entrada del Lector:** Adquiere P(e). Si hay escritores activos, se incrementa dr y se bloquea en P(r).
- **Entrada del Escritor:** Adquiere P(e). Si hay lectores o escritores activos, se incrementa dw y se bloquea en P(w).
- **Salida (SIGNAL):** Verifica condiciones para despertar lectores o escritores según corresponda.

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

La instrucción AWAIT es una herramienta en programación concurrente que permite que un proceso se detenga hasta que se cumpla una condición específica. Su sintaxis es:

Semántica de AWAIT:

- La condición B debe ser verdadera antes de que se ejecute la secuencia S. Mientras B sea falsa, el proceso se bloqueará, permitiendo que otros procesos continúen. Una vez que B se cumple, el proceso se reanuda y ejecuta S de manera atómica, evitando interferencias.

Relación con TEST & SET (TS) y FETCH & ADD (FA):

- **TEST & SET:** Esta instrucción atómica modifica y verifica el valor de una variable booleana, implementando spin locks que mantienen a un proceso en espera mientras una sección crítica está ocupada.
- **FETCH & ADD:** Esta instrucción atómica incrementa una variable y devuelve su valor anterior, útil en la sincronización de procesos en estructuras como barreras.

Implementación de AWAIT:

- AWAIT puede implementarse usando TS o FA, creando una sección crítica alrededor de la verificación de la condición B y la ejecución de S.
- **Ejemplo con TEST & SET:**

Definición de Concurrencia y Diferencias con el Procesamiento Secuencial y Paralelo

La **concurrencia** se refiere a la capacidad de ejecutar múltiples actividades simultáneamente, lo que permite que distintos objetos actúen al mismo tiempo. Este

concepto es fundamental en el diseño de hardware, sistemas operativos, multiprocesadores y programación.

Se pueden diferenciar tres tipos de procesamiento:

1. **Procesamiento secuencial**: Solo hay un flujo de control, ejecutando instrucciones una a la vez en un orden estricto. Un ejemplo sería la fabricación de un objeto en una sola máquina, donde cada parte se produce de forma consecutiva.
2. **Procesamiento concurrente**: Involucra múltiples hilos que se ejecutan "al mismo tiempo", aunque no necesariamente en paralelo. Esto puede suceder en un único procesador donde el sistema operativo comparte el tiempo de CPU. En la fabricación, sería como si una máquina dedicara un tiempo parcial a cada componente.
3. **Procesamiento paralelo**: Se refiere a la ejecución concurrente en múltiples procesadores, con el objetivo de reducir el tiempo de ejecución. En el ejemplo de fabricación, sería tener una máquina para cada parte, trabajando al mismo tiempo.

Tipos de Fairness en Programación Concurrente

La **fairness** en programación concurrente se refiere a la garantía de que todos los procesos tengan la oportunidad de progresar, independientemente del comportamiento de los demás. Existen tres tipos principales:

1. **Fairness Incondicional**: Asegura que cualquier acción atómica elegible se ejecutará eventualmente. Un ejemplo es la política Round Robin en monoprocesadores, donde cada proceso tiene un turno garantizado.
2. **Fairness Débil**: Además de garantizar la ejecución de acciones incondicionales, permite que una acción atómica condicional se ejecute si su condición se vuelve verdadera y permanece así. Sin embargo, no garantiza la ejecución si la condición cambia mientras el proceso espera.
3. **Fairness Fuerte**: Es la forma más robusta de fairness. Garantiza la ejecución de acciones incondicionales y asegura que las acciones condicionales se ejecuten si su condición se vuelve verdadera infinitas veces, independientemente de los cambios en el estado de la condición.

Monitores y semáforos

Semáforos

- **Definición:** Variables protegidas que controlan el acceso a recursos compartidos por múltiples procesos.
- **Propiedades:**
    - **Valor entero no negativo:** Representa la cantidad de recursos disponibles.
    - **Operaciones atómicas:**

        - **P (espera o decremento):** Disminuye el valor del semáforo; bloquea el proceso si el valor es negativo.
        - **V (señal o incremento):** Aumenta el valor del semáforo; desbloquea procesos en espera.
- **Tipos:**
    - **Semáforo binario:** Solo valores 0 y 1; garantiza exclusión mutua.
    - **Semáforo de conteo:** Puede ser cualquier valor entero no negativo; controla acceso a grupos de recursos idénticos.
- **Usos comunes:**
    - Exclusión mutua.
    - Sincronización condicional.
    - Control de concurrencia.
- **Limitaciones:**
    - Complejidad en el uso y propensión a errores.
    - Dificultad de depuración en programas grandes.

Monitores

- **Definición:** Construcciones de alto nivel que gestionan la concurrencia de manera estructurada.
- **Propiedades:**
    - **Exclusión mutua implícita:** Solo un proceso ejecuta un procedimiento de monitor a la vez.
    - **Variables de condición:** Permiten a los procesos esperar condiciones específicas.
    - **Operaciones de señalización:** Incluyen wait, signal, y signal_all para gestionar procesos bloqueados.
- **Beneficios:**
    - **Abstracción:** Facilita el desarrollo al ocultar la complejidad de la sincronización.
    - **Modularidad:** Encapsula datos y procedimientos en una unidad.
    - **Facilidad de mantenimiento:** Programas más comprensibles y fáciles de mantener.
- **Implementación:** Puede hacerse a nivel de sistema operativo o usando mecanismos de sincronización de bajo nivel, como semáforos.

Sincronización

Definición: La sincronización es la coordinación de acciones entre procesos o hilos que se ejecutan simultáneamente. Es esencial para garantizar la corrección de los programas concurrentes, previniendo condiciones de carrera y asegurando que los datos compartidos se acceden y modifican de manera ordenada y coherente. Esto restringe las historias posibles de un programa concurrente a aquellas que son deseables, ya que diferentes interleavings de instrucciones pueden dar lugar a resultados incorrectos.

Tipos de Sincronización

1. **Sincronización por Exclusión Mutua:**
    - **Objetivo:** Asegurar que solo un proceso acceda a un recurso compartido a la vez (como variables, archivos o dispositivos).
    - **Mecanismos:**
        - **Secciones Críticas:** Bloques de código que deben ejecutarse sin interferencia de otros procesos.
        - **Locks (Cerraduras):** Variables que un proceso debe adquirir antes de entrar en una sección crítica y liberar al salir. Ejemplos incluyen spin locks, algoritmos como Tie-Breaker, y estructuras como semáforos y monitores.
    - **Ejemplo:** En un sistema de reservas de vuelos, la exclusión mutua asegura que dos usuarios no puedan reservar el mismo asiento simultáneamente.
2. **Sincronización por Condición:**
    - **Objetivo:** Permitir que un proceso espere a que se cumpla una condición específica antes de continuar su ejecución, dependiendo del estado de los datos compartidos.
    - **Mecanismos:**
        - **Variables de Condición:** Estructuras dentro de monitores que permiten a los procesos esperar (usando `wait`) y señalar (usando `signal` o `signal_all`) el cumplimiento de una condición.
        - **Semáforos:** También pueden utilizarse para implementar la sincronización condicional.
    - **Ejemplo:** En un sistema productor-consumidor, un consumidor puede esperar a que un productor agregue datos a un búfer antes de intentar consumirlos.

Comunicación

Definición: La comunicación entre procesos se refiere al mecanismo por el cual los procesos o hilos que se ejecutan simultáneamente intercambian información. Este aspecto es fundamental en la programación concurrente, ya que está estrechamente relacionado con la sincronización y la coordinación del intercambio de datos con el acceso a recursos compartidos.

Paradigmas Principales de Comunicación

1. **Memoria Compartida (MC):**
    - **Descripción:** Los procesos se comunican accediendo a áreas de memoria compartidas, donde pueden leer y escribir datos directamente sin necesidad de enviar mensajes explícitos.
    - **Ventajas:**

        - Alta eficiencia, especialmente en arquitecturas de memoria compartida, al evitar la sobrecarga del envío de mensajes.
    - **Desafíos:**
        - **Exclusión Mutua:** Es fundamental garantizar que solo un proceso modifique los datos compartidos a la vez, utilizando mecanismos de sincronización como semáforos o monitores.
        - **Coherencia de Datos:** En sistemas de memoria distribuida, se requieren protocolos de coherencia para asegurar que todos los procesos tengan una vista consistente de los datos.
2. **Pasaje de Mensajes (PM):**
    - **Descripción:** Los procesos se comunican enviando mensajes explícitos a través de canales de comunicación, que pueden ser lógicos (como colas de mensajes) o físicos (como enlaces de red).
    - **Ventajas:**
        - **Flexibilidad:** Se adapta bien a diferentes arquitecturas, incluyendo sistemas de memoria distribuida y entornos de red.
        - **Modularidad:** Facilita el diseño modular, ya que los procesos no necesitan compartir memoria directamente, reduciendo el acoplamiento.
    - **Desafíos:**
        - **Sobrecarga:** Enviar mensajes puede tener una mayor sobrecarga en comparación con el acceso a la memoria compartida, especialmente en sistemas con alta latencia de comunicación.
        - **Complejidad de Sincronización:** Aunque puede simplificar la exclusión mutua, requiere mecanismos de sincronización para coordinar el envío y la recepción de mensajes, utilizando mensajes sincrónicos o asincrónicos, cada uno con sus ventajas y desventajas.