¿Qué es la concurrencia?

La concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente. Permite a distintos objetos actuar al mismo tiempo

Ejemplos concurrencia:

- Navegador Web accediendo una página mientras atiende al usuario.
- Varios navegadores accediendo a la misma página.
- Acceso a disco mientras otras aplicaciones siguen funcionando
- Los sistemas biológicos

Procesamiento secuencial

El procesamiento secuencial implica realizar una tarea o proceso a la vez, siguiendo un orden temporal estricto. Cada tarea debe esperar a que la anterior termine antes de comenzar su ejecución. Hay un único flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente.

Procesamiento concurrente

El procesamiento concurrente se refiere a la ejecución de múltiples tareas o procesos de manera aparentemente simultánea, pero no necesariamente en paralelo.

Puede llevarse a cabo en sistemas de un solo procesador mediante la alternancia rápida entre tareas o en sistemas con múltiples procesadores. No está restringido a una arquitectura particular de hardware ni a un número determinado de procesadores. Especificar la concurrencia implica especificar los procesos **concurrentes**, su **comunicación** y su **sincronización**.

La concurrencia sin paralelismo es la multiprogramación en un procesador en donde el tiempo de CPU es compartido entre varios procesos, por ejemplo por time slicing. El sistema operativo controla y planifica procesos: si el slice expiró o el proceso se bloquea el sistema operativo hace context (process) switch.

El procesamiento paralelo es la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.

Un programa concurrente especifica dos o más “programas secuenciales” que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos. Un proceso o tarea es un elemento concurrente abstracto que puede ejecutarse simultáneamente con otros procesos o tareas, si el hardware lo permite.

Un programa concurrente puede tener N procesos habilitados para ejecutarse concurrentemente y un sistema concurrente puede disponer de M procesadores cada uno de los cuales puede ejecutar uno o más procesos.

- Procesos: Cada proceso tiene su propio espacio de direcciones y recursos.
- Procesos livianos, threads o hilos:

    - Proceso “liviano” que tiene su propio contador de programa y su pila de ejecución, pero no controla el “contexto pesado” (por ejemplo, las tablas de página).
    - Todos los hilos de un proceso comparten el mismo espacio de direcciones y recursos (los del proceso).
    - El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias.
    - La concurrencia puede estar provista por el lenguaje o por el Sistema Operativo

Los procesos cooperan y compiten (o son totalmente independientes, muy raro)

Objetivos de los sistemas concurrentes

- Ajustar el modelo de arquitectura de hardware y software al problema del mundo real a resolver
- Incrementar la performance, mejorando los tiempos de respuesta de los sistemas de cómputo, a través de un enfoque diferente de la arquitectura física y lógica de las soluciones.
- Algunas ventajas:
    - La velocidad de ejecución que se puede alcanzar.
    - Mejor utilización de la CPU de cada procesador.
    - Explotación de la concurrencia inherente a la mayoría de los problemas reales.

Administración recursos compartidos

Esto incluye la asignación de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.

Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (fairness).

Dos situaciones NO deseadas en los programas concurrentes son la inanición de un proceso (no logra acceder a los recursos compartidos) y el overloading de un proceso (la carga asignada excede su capacidad de procesamiento).

Otro problema importante que se debe evitar es el deadlock.

Competencia

Típico en Sistemas Operativos y Redes, debido a recursos compartidos.

Pueden generar deadlock o inanición

Deadlock

Un deadlock (bloqueo) es una situación en la que dos o más procesos o hilos en un sistema concurrente quedan atrapados en un estado en el que ninguno puede

continuar su ejecución debido a que cada uno está esperando que el otro libere un recurso que necesita. Esto puede resultar en un estancamiento completo del sistema.

Condiciones para que ocurra:

1. Recursos reusables serialmente: los procesos comparten recursos que pueden usar con exclusión mutua.
2. Adquisición incremental: los procesos mantienen los recursos que poseen mientras esperar adquirir recursos adicionales.
3. No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.
4. Espera cíclica: existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.

Con evitar que se cumpla alguna de las condiciones mencionadas, se evita el deadlock.

Inanición

La inanición es una situación en la que un proceso o hilo en un sistema concurrente no puede avanzar o realizar su trabajo debido a la incapacidad de acceder a los recursos compartidos o a las condiciones necesarias para su ejecución. Esto puede deberse a que otros procesos tienen prioridad sobre él y constantemente obtienen acceso a los recursos, dejando al proceso en estado de espera indefinida.

Ejemplo:

En un sistema operativo con múltiples procesos compitiendo por el acceso a una impresora compartida. Cada proceso necesita imprimir un documento en la impresora. Si se implementa una política de asignación de recursos que siempre otorga acceso a la impresora al mismo grupo de procesos o al proceso de mayor prioridad, otros procesos con menor prioridad pueden quedar en estado de inanición.

Si tenemos tres procesos: A, B y C, donde A y B tienen prioridad alta y C tiene prioridad baja, y la política de asignación de recursos da preferencia a los procesos de alta prioridad, entonces el proceso C puede quedarse esperando indefinidamente para imprimir su documento.

No determinismo

En el no determinismo no hay un orden preestablecido en la ejecución, la ejecución de esta “entrada” puede generar diferentes “salidas”.

Cuando se aplica este concepto a la ejecución concurrente, significa que en un entorno con múltiples hilos o procesos en ejecución, el orden en que se ejecutan las operaciones o eventos no está preestablecido y puede variar de una ejecución a otra, incluso si las entradas y las condiciones iniciales son las mismas.

Comunicación entre procesos

La comunicación entre procesos concurrentes indica el modo en que se organizan y trasmiten datos entre tareas concurrentes. Esta organización requiere especificar protocolos para controlar el progreso y la corrección.

Los procesos se comunican:

- Por Memoria Compartida.
    - Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella.
    - Lógicamente no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a bloquear y liberar el acceso a la memoria.
    - La solución más elemental es una variable de control tipo “semáforo” que habilite o no el acceso de un proceso a la memoria compartida.
    - Todos los procesos acceden al mismo espacio de direcciones
- Por Pasaje de Mensajes
    - Es necesario establecer un canal (lógico o físico) para transmitir información entre procesos.
    - También el lenguaje debe proveer un protocolo adecuado.
    - Para que la comunicación sea efectiva los procesos deben “saber” cuándo tienen mensajes para leer y cuando deben trasmitir mensajes.

Puede haber:

Máquinas de memoria compartida

Interacción modificando datos en la memoria compartida.

- Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problemas de sincronización y consistencia.
    - Todas las unidades de procesamiento tardan el mismo tiempo para acceder a la memoria (acceden a través del bus)
- Esquemas NUMA para mayor número de procesadores distribuidos.
    - Cada unidad tiene cierto bloque de memoria. Si una unidad quiere acceder a la memoria de otro bloque, tarda mas.
- Problema de consistencia.

Memoria distribuida

Procesadores conectados por una red.

- Memoria local (no hay problemas de consistencia).
- Interacción es sólo por pasaje de mensajes.
- Grado de acoplamiento de los procesadores:
    - Multicomputadores (máquinas fuertemente acopladas). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.
    - Memoria compartida distribuida.

    - Clusters.
    - Redes (multiprocesador débilmente acoplado).

Sincronización

La sincronización es la posesión de información acerca de otro proceso para coordinar actividades.

Los procesos se sincronizan:

- Por exclusión mutua.
    - Asegura que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
    - Si el programa tiene secciones críticas que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.
- Por condición.
    - Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

Pueden estar presentes más de un mecanismo de sincronización. Un ejemplo de esto es el problema de un "buffer limitado" compartido entre productores y consumidores. En este escenario, se utilizan mecanismos de exclusión mutua para garantizar que un productor o consumidor acceda al búfer de manera exclusiva cuando lo necesita. A su vez, se utilizan mecanismos de condición para permitir que los productores esperen si el búfer está lleno o que los consumidores esperen si el búfer está vacío, asegurando así que la sincronización se realice correctamente para mantener la integridad de los datos y evitar bloqueos o inanición.

Granularidad

La granularidad de una aplicación está dada por la relación entre el cómputo y la comunicación.

- Relación y adaptación a la arquitectura.
- Grano fino y grano grueso
    - Poco código, comunicación → Grano fino
    - Mucho código, comunicación → Grano grueso.

Requerimientos de un lenguaje de programación concurrente:

Independientemente del mecanismo de comunicación / sincronización entre procesos, los lenguajes de programación concurrente deberán proveer primitivas adecuadas para la especificación e implementación de las misma

- Indicar las tareas o procesos que pueden ejecutarse concurrentemente.
- Mecanismos de sincronización.
- Mecanismos de comunicación entre los procesos.

Problemas

1. Complejidad de Exclusión Mutua y Sincronización:
    a. La necesidad de gestionar la exclusión mutua y sincronización agrega complejidad al diseño de programas concurrentes.
2. Procesos No "Vivos" en Programas Concurrentes:
    a. Los procesos iniciados en programas concurrentes pueden no mantener su "liveness", lo que puede indicar deadlocks o una asignación ineficiente de recursos.
3. No Determinismo en Interleaving de Procesos:
    a. El interleaving de procesos concurrentes introduce no determinismo, lo que implica que dos ejecuciones del mismo programa no son necesariamente idénticas. Esto complica la interpretación y depuración del código.
4. Reducción de Performance por Overhead:
    a. Existe la posibilidad de una disminución en el rendimiento debido al overhead asociado con cambios de contexto, comunicación entre procesos, sincronización, etc.
5. Mayor Tiempo de Desarrollo y Puesta a Punto:
    a. El desarrollo de software concurrente y la optimización para la ejecución paralela pueden requerir más tiempo en comparación con enfoques secuenciales. Además, la paralelización de algoritmos secuenciales puede ser difícil.
6. Adaptación del Software al Hardware Paralelo:
    a. Para lograr mejoras reales en el rendimiento, es necesario adaptar el software concurrente al hardware paralelo subyacente. Esto añade otra capa de complejidad al proceso de desarrollo.

Interferencia

La interferencia es una situación en la que un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

Para evitar la interferencia en programación concurrente se utilizan las acciones atómicas y técnicas de sincronización. Las operaciones atómicas son aquellas que se ejecutan como una única unidad indivisible, lo que significa que no pueden ser interrumpidas por otros procesos. Esto garantiza que, cuando un proceso está realizando una operación atómica en un recurso compartido, ningún otro proceso puede interferir en medio de esa operación.

Prioridad

Un proceso que tiene mayor prioridad puede causar la suspensión (preemption) de otro proceso concurrente. Análogamente puede tomar un recurso compartido, obligando a retirarse a otro proceso que lo tenga en un instante dado.

La granularidad de una aplicación está dada por la relación entre el cómputo y la comunicación. Relación y adaptación a la arquitectura. Grano fino y grano grueso

Atomicidad

Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas. Una acción atómica hace una transformación de estado indivisibles (estados intermedios invisibles para otros procesos)

En la ejecución de un programa concurrente hay un intercalado de las acciones atómicas ejecutadas por procesos individuales..

El número de posibles historias (trace) de un programa concurrente es enorme; pero no todas son válidas

Como algunas historias son válidas y otras no se debe asegurar un orden temporal entre las acciones que ejecutan los procesos, es decir, como las tareas se intercalan. Para ello, deben fijarse restricciones.

Atomicidad de grano fino

Una acción atómica de grano fino (fine grained) se debe implementar por hardware.

Si una expresión e en un proceso no referencia una variable alterada por otro proceso, la evaluación será atómica, aunque requiera ejecutar varias acciones atómicas de grano fino

Si una asignación x = e en un proceso no referencia ninguna variable alterada por otro proceso, la ejecución de la asignación será atómica.

Referencia crítica

En una expresión, es la referencia a una variable que es modificada por otro proceso. Se asume que toda referencia crítica es a una variable simple leída y escrita atómicamente.

Propiedad de “A lo sumo una vez”

Una sentencia de asignación x = e satisface la propiedad de “A lo sumo una vez” si:

1) e contiene a lo sumo una referencia crítica y x no es referenciada por otro proceso, o
2) e no contiene referencias críticas, en cuyo caso x puede ser leída por otro proceso.

Una expresiones e que no está en una sentencia de asignación satisface la propiedad de “A lo sumo una vez” si no contiene más de una referencia crítica.

Básicamente la sentencia una sentencia cumple ASV si hay a lo sumo una variable compartida, y esta tiene que ser referenciada a lo sumo una vez

Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución parece atómica, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos:

- int x=0, y=0;
  co x=x+1 // y=y+1 oc;
  No hay ref. críticas en ningún proceso.
  En todas las historias ***x = 1*** e ***y = 1***

- int x = 0, y = 0;
  co x=y+1 // y=y+1 oc;
  El 1er proceso tiene 1 ref. crítica. El 2do ninguna.
  Siempre ***y = 1*** y ***x = 1 o 2***

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

Dado el siguiente programa concurrente:

```
x = 2; y = 4; z = 3;
 co
   x = y - z // z = x * 2 // y = y - 1
oc
```

a) *¿Cuáles de las asignaciones dentro de la sentencia co cumplen con ASV?. Justifique claramente.*

x = y – z no cumple porque se están usando 3 variables compartidas
*z = x * 2 no cumple porque se están usando 2 variables compartidas*
y = y -1 cumple porque se esta usando una variable compartida y es referenciada una vez

b) *Indique los resultados posibles de la ejecución*
Nota 1: las instrucciones NO SON atómicas.
Nota 2: no es necesario que liste TODOS los resultados, pero si los que sean representativos de las diferentes situaciones que pueden darse.

X = Y – 1

1) *Load posMemY, acum*
2) *Sub 2, acum*
3) *Store acum, posMemX*

*Z = X * 2*

4) *Load posMemX, acum*
5) *Add posMemX, acum*
6) *Store acum, posMemZ*

Y = Y – 1

7) *Load posMemY, acum*
8) *Sub 1, acum*
9) *Store acum, posMemY*

Resultados posibles

1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 → Z = 4, X = 2, Y = 3

1 – 2 – 4 – 5 – 6 – 3 – 7 – 8 – 9 → Z = 4, X = 2, Y = 3

7 – 8 – 9 – 1 – 2 – 3 – 4 – 5 – 6 → Z = 2, X = 1, Y = 3

7 – 8 – 9 – 1 – 2 – 4 – 5 – 3 – 6 → Z = 4, X = 1, Y = 3

4 – 7 – 8 – 9 – 1 – 2 – 3 – 5 – 6 → Z = 3, X= 1, Y = 3

Acciones atómicas condicionales e incondicionales

Acciones atómicas incondicionales: son acciones que se ejecutan sin considerar ninguna condición previa. Siempre se realizan de manera atómica, independientemente de las circunstancias. < x = 1 >

Acciones atómicas condicionales: son acciones atómicas que se ejecutan cuándo se cumple una cierta condición previa. < await S; x = 1 > (puede ser solo < await S >)

Propiedades de seguridad y vida

Seguridad (safety)

- Esta propiedad se refiere a la garantía de que "nada malo le ocurre a un proceso" o, en otras palabras, que no se producen estados inconsistentes o resultados incorrectos debido a la ejecución concurrente.
    - Una falla de seguridad indica que algo anda mal.
    - Ejemplos de propiedades de seguridad: exclusión mutua, ausencia de interferencia entre procesos, partial correctness.

Vida (liveness)

- Esta propiedad se relaciona con la garantía de que "eventualmente ocurre algo bueno con una actividad". Se enfoca en asegurarse de que los procesos no queden bloqueados (deadlocks) y que las actividades puedan avanzar y completarse.
    - Una falla de vida indica que las cosas dejan de ejecutar.
    - Ejemplos de vida: terminación, asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, etc ⇒ dependen de las políticas de scheduling

Fairness

Una política de scheduling se refiere a un conjunto de reglas y algoritmos utilizados para determinar cuál será el próximo proceso o hilo que se ejecutará en un sistema

concurrente o en un sistema operativo. Cuando se habla de ejecutar el próximo proceso o hilo, se esta hablando de ejecutar las acciones atómicas de dicho proceso.

Fairness se relaciona con la equidad en la ejecución de procesos o hilos en un sistema concurrente. Se esfuerza por garantizar que todos los procesos tengan la oportunidad de avanzar y realizar sus operaciones, evitando situaciones de bloqueo o inanición.

Fairness trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás, es decir, que se ejecuten acciones atómicas de todos lo procesos.

Tipos de Fairness (varias definiciones porque soy tonta):

Fairness Incondicional: Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

Fairness Débil: Una política de scheduling es débilmente fair si :

(1) Es incondicionalmente fair y

(2) Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece true hasta que es vista por el proceso que ejecuta la acción atómica condicional.

Fairness Fuerte: Una política de scheduling es fuertemente fair si:

(1) Es incondicionalmente fair y

(2) Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

Explicación de chatGPT porque no las entendí mucho:

Fairness Incondicional: Esta política garantiza que toda acción atómica elegible eventualmente se ejecute, sin importar las condiciones o restricciones. Se enfoca en garantizar que todas las acciones tengan una oportunidad justa de ejecutarse.

Fairness Débil: Además de ser incondicionalmente justa, esta política también garantiza que las acciones atómicas condicionales que se vuelven elegibles eventualmente se ejecuten siempre que su condición se cumpla. Esto evita que las acciones condicionales queden en espera indefinidamente si su condición se cumple.

Fairness Fuerte: Es la política más exigente en términos de equidad. Además de ser incondicionalmente justa, garantiza que las acciones atómicas condicionales se ejecuten con una frecuencia infinita si su condición se cumple. Esto evita que las acciones condicionales queden en espera incluso si su condición se cumple raramente.

Fairness Incondicional:

Garantiza que toda acción atómica elegible eventualmente se ejecute, independientemente de las condiciones o restricciones.

Se centra en proporcionar una oportunidad justa para todas las acciones sin importar sus condiciones.

No tiene en cuenta si las condiciones de las acciones condicionales se cumplen o no, todas las acciones elegibles se ejecutarán en algún momento.

Fairness Débil:

Además de ser incondicionalmente justa, también se preocupa por las acciones condicionales.

Garantiza que las acciones atómicas condicionales que se vuelven elegibles eventualmente se ejecuten siempre que su condición se cumpla.

Evita que las acciones condicionales queden en espera indefinidamente si su condición se cumple.

Fairness Fuerte:

La política más exigente en términos de equidad.

Además de ser incondicionalmente justa, garantiza que las acciones atómicas condicionales se ejecuten con una frecuencia infinita si su condición se cumple.

Evita que las acciones condicionales queden en espera, incluso si su condición se cumple raramente.

Fairness Incondicional. Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

Fairness Débil. Una política de scheduling es débilmente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda se convierte en true y de allí en adelante permanece true.

Fairness Fuerte. Una política de scheduling es fuertemente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda es true con infinita frecuencia.

Fairness incondicional: significa que todas las tareas que están listas para ejecutarse eventualmente se ejecutan, sin importar si son condicionales o no.

Fairness débil: significa que además de la fairness incondicional, las tareas condicionales que se vuelven elegibles se ejecutan, siempre y cuando su condición se mantenga verdadera hasta que la tarea sea asignada.

Fairness fuerte: significa que además de la fairness incondicional, las tareas condicionales que se vuelven elegibles se ejecutan, incluso si su condición cambia de valor muchas veces.