## Programación Concurrente

**1-Conseptos Básicos:**

¿Qué es la concurrencia? Es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente. Es un concepto de software no restringido a una arquitectura particular de hardware ni a un numero determinado de procesadores.

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 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.

Procesamiento secuencia, concurrente y paralelo:

**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.

Características importantes:

Interacción.

No Determinismo: puede dar distintos resultados al ejecutarse sobre los mismos datos de entrada.

La concurrencia no implica paralelismo.

**Concurrencia “Interleaved”:** Proceso lógicamente simultáneo. Ejecución intercala en un único procesador. Pseudo-Paralelismo.

**Concurrencia Simultánea:** Procesamiento físicamente simultaneo. Requiere un sistema multiprocesador o multicore. Paralelismo Full.

Programa secuencial: Totalmente ordenado, determinantico (para los mismo datos de entrada, ejecuta siempre la misma secuencia de instrucciones y obtiene la misma salida)

* La Concurrencia es un concepto de software. En Programación Concurrente la organización de ***procesos y procesadores*** constituyen la arquitectura del sistema concurrente.
* La Programación Paralela se asocia con la ejecución concurrente en múltiples procesadores que pueden tener memoria compartida, y generalmente con un objetivo de incrementar performance.
* La Programación Distribuida es un “caso” de concurrencia con múltiples procesadores y sin memoria compartida.

***Especificar la concurrencia es esencialmente especificar los procesos concurrentes, su comunicación y sincronización.***

Evolución Historia:

60’: Evolucion del SO, no determinismo, problema de la SC.

70’: Formalizacion de concurrencia en los lenguales.

80´: Redes, procesamiento distribuido.

90’: MPP, internet, Cliente/servidor, web compiuting

2000’: procesamiento masivo de datos distribuidos, clusters, computación móvil.

Procesos. Programa concurrente. No determinismo.

Proceso secuencial: un solo flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente.

Una solución secuencial nos fuerza a tener un orden temporal estricto.

Con N maquinas podemos trabajar al mismo tiempo si no hay dep de recursos.

Procesos e hilos:

Todos los sistemas operativos soportan procesos. Cada proceso se ejecuta en una máquina virtual distinta.

Algunos sistemas operativos soportan procesos ligeros (hilos o threads)

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 la misma máquina virtual. Tienen acceso al mismo espacio de memoria.

El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias.

La concurrencia puede estar soportada por el lenguaje: Java, Ada, occam2, el SO: C/posix.

No determinismo: Los programas concurrentes suelen ser no deterministicos puede dar resultados distintos al ejecutarse sobre los mismo datos de entrada.

Clases de aplicaciones. Multithreading, Cómputo paralelo y distribuido.

**Multithreading - Ejecución de N procesos independientes en M procesadores (N>M):** Un sistema de software de “multithreading” maneja simultáneamente tareas independientes, asignando los procesadores de acuerdo a alguna política (ej, por tiempos).

Organización más “natural” como un programa concurrente.

**Cómputo distribuido**: Una red de comunicaciones vincula procesadores diferentes sobre los que se ejecutan procesos que se comunican esencialmente por mensajes.

Cada componente del sistema distribuido puede hacer a su vez multithreading.

**Procesamiento paralelo**: Resolver un problema en el menor tiempo (o un problema + grande en aprox. el mismo tiempo) usando una arq. Multiprocesador en la que se pueda distribuir la tarea global en tareas (independientes? interdependientes?) que puedan ejecutarse en != procesadores.

Paralelismo de datos y paralelismo de procesos.

Concurrencia y paralelismo.

Un programa concurrente puede ser ejecutado por:

**Multiprogramación:** los procesos comparten uno o más procesadores

**Multiprocesamiento:** cada proceso corre en su propio procesador pero con memoria compartida

**Procesamiento Distribuido:** cada proceso corre en su propio procesador conectado a los otros a través de una red

Algoritmos concurrentes, distribuidos y paralelos.

Áreas de estudio en sistemas concurrentes.

Relación con la arquitectura. Monoprocesadores. Multiprocesadores: Clasificaciones y ejemplos. Conceptos de arquitecturas Grid. Memoria compartida distribuida.

Concurrencia a nivel hardware: Maquinas monoprocesador, ya no pueden mejorar.

**Multiprocesadores de memoria compartida.**

La interacción se da modificando datos almacenados en la MC

- Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problema de sincronización y *consistencia*

- Esquemas NUMA para mayor número de procesadores distribuidos.

- Problema de consistencia

**Multiprocesadores con memoria distribuida.**

Procesadores conectados por una red. C/u tiene memoria local y la interacción es sólo por pasaje de mensajes.

Grado de acoplamiento de los procesadores:

- Multicomputadores *(tightly coupled machine*). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.

- Redes (*loosely coupled multiprocessor*)*.*

- NOWs / Clusters.

- Memoria compartida distribuida.

Relación con el sistema operativo. Requerimientos para el sistema operativo.

Relación con el lenguaje. Requerimientos para el sistema operativo.

Sincronización y comunicación.

Los procesos se COMUNICAN. *La* ***comunicación*** *indica el modo en que se organiza y trasmiten datos entre tareas concurrentes.* Esta organización requiere especificar *protocolos*para controlar el progreso y corrección de la comunicación, por memoria compartida o por pasaje de mensajes.

**Sincronización** es la posesión de información acerca de otro proceso para coordinar actividades. Es un estado de un programa concurrente. **El objetivo de la sincronización es restringir las historias de un programa concurrente solo a las permitidas.**

Sincronización por exclusión mutua y por condición.

Sincronización por exclusión mutua: asegura que solo un proceso tenga acceso a un recurso compartido en un instante de tiempo. Si el proceso tiene secciones críticas que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrase en la misma sección critica al mismo tiempo.

Sincronización por condición: permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

Comunicación por memoria compartida y por mensajes.

**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 MC, lo que obliga a bloquear y liberar el acceso a la memoria.

**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.

Prioridad, granularidad, deadlock, manejo de recursos:

Un proceso que tiene mayor **prioridad** puede causar la suspensión 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. Relación entre computo y comunicación.

Para una aplicación dada, significa optimizar la relación entre el numero de procesadores y el tamaño de memoria total.

Clasificación por la **granularidad de los procesadores**:

**De grano grueso** (*coarse-grained*): pocos procesadores muy poderosos.

**De grano fino** (*fine-grained*): gran número de procesadores menos potentes.

**De grano medio** (*medium-grained*).

**Manejo de recursos:** uno de los temas principales de la programación concurrente es la administración de 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(a un proceso se le asigna un recurso que implican mas trabajo que el de otro).

**Deadlock:** Dos (o mas) procesos pueden entrar en deadlock, si por error de programación ambos se quedan esperando que el otro libere un recurso compartido. La ausencia de deadlock es una propiedad necesaria en los procesos concurrentes.

4 propiedades necesarias y suficientes para que exista deadlock son:

-Recursos reusables seriamente: los procesos comparten recursos que pueden usar con exclusión mutua.

-Adquisición incremental: los proceso mantienen los recursos que poseen mientras esperan adquirir recursos adicionales.

-No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que solo son liberados voluntariamente.

-Espera critica: existe una cadena circular de procesos tal que cada uno tiene un recurso que su sucesor en el cielo está esperando adquirir.

Paradigmas de resolución de programas concurrentes:

Iterativo (ej, multiplicación de matrices):

En el paralelismo interactivo un programa consta de un conjunto de procesos (posiblemente idénticos) cada uno de los cuales tiene 1 o mas loops. Cada proceso es un programa interativo.

Los procesos cooperan para resolver un único problema, pueden trabajar independientemente, y comunicarse y sincronizar por memoria compartida o pasaje de mensajes.

Recursivo o *divide & conquer* (ej. el problema de la cuadratura):

En el paralelismo recursivo el problema general /programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos(dividir y conquistar).

*Pipeline* o productor consumidor:

Los esquemas *productor-consumidor* muestran procesos que se comunican. Es habitual que estos procesos se organicen en pipes a través de los cuales fluye la información. Cada proceso en el pipe es un filtro que consume la salida de su proceso predecesor y produce una salida para el proceso siguiente. Ejemplos a distintos niveles de SO.

Cliente/servidor y sus variantes:

*Cliente-servidor* es el esquema dominante en las aplicaciones de procesamiento distribuido. Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Unos y otros pueden ejecutarse en procesadores diferentes. Comunicación bidireccional. Atención de a un cliente o con multithreading a varios. Mecanismos de invocación variados (rendezvous y RPC x ej en MD, monitores x ej en MC). El soporte distribuido puede ser simple (LAN) o extendido a la WEB.

*Peers* o pares que interactúan:

En los esquemas de *pares que interactúan* los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea y completar el objetivo. Permite mayor grado de asincronismo que C/S

Configuraciones posibles: grilla, pipe circular, uno a uno, arbitraria

**2. *Concurrencia y sincronización***

Aspectos de programación secuencial:

Cada proceso concurrente es un programa secuencial. La programación secuencial estructurada puede expresarse con 3 clases de instrucciones básicas: asignación, alternativa(decisión) e iteración.

Especificación y semántica de la ejecución concurrente. La sentencia *co* y *process*

Sentencia ***co***:

**co S1 // ..... // Sn oc →** Ejecuta las *Si* tareas concurrentemente.

La ejecución del *co* termina cuando todas las tareas terminaron.

Cuantificadores:

**co [i=1 to n] { a[i] = 0; b[i] = 0 } oc →** Crea *n* tareas concurrentes.

***Process:*** otra forma de representar concurrencia

**process A {sentencias} →** proceso único independiente.

Cuantificadores

**process B [i=1 to n] {sentencias} →** *n* procesos independientes.

**Diferencia: *process*** ejecuta en ***background***, mientras el código que contiene un ***co*** espera a que el proceso creado por la sentencia ***co*** termine antes de ejecutar la siguiente sentencia.

Acciones atómicas y sincronización:

Estado de un programa concurrente.

Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o mas acciones atómicas.

Una acción atómica hace una transformación de estado indivisibles. La ejecución de un programa concurrente es intercalada (interleaving) de acciones atómicas ejecutadas por procesos individuales.

Interacción-> no todos los interleavings son aceptables.

Historia de un programa concurrente (trance).

El objetivo de la sincronización es prevenir los interleaving indeseables restringiendo las historias de un programa concurrente solo a las permitidas.

El problema de interferencia. Historias válidas e inválidas:

Interferencia: un proceso toma un acción que invalida las suposiciones hechas por otro proceso.

“Interleaving extremo”: Dos procesos que realizan (c/u) *k* iteraciones de la sentencia N=N+1 (N compartida init 0)

Process P1{ Process P2{ fa i=1 to K  N=N+1 af fa i=1 to K  N=N+1 af } }

¿Cuál puede ser el valor final de N?

- 2K

- entre K+1 y 2K-1

- K

- <K (incluso 2…)

***Cuándo valdrá 2?***

1. Proceso 1: Load N

2. Proceso 2: Hace k-1 iteraciones del loop

3. Proceso 1: Incrementa su copia

4. Proceso 1: Store N

5. Proceso 2: Load N

6. Proceso 1: Hace k-1 iteraciones del loop

7. Proceso 2: Incrementa su copia

8. Proceso 2: Store N

Atomicidad de grano fino y de grano grueso:

La propiedad de “a lo sumo una vez”.

*Referencia crítica* en una expresión -> referencia a una variable que es modificada por otro proceso. Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

*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*

**Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez.**

Una def. similar se aplica a expresiones que no están en sentencias de asignación (satisface ASV si no contiene más de una ref. crítica)

La sentencia Await. Semántica. Especificación de la sincronización.

Técnicas para evitar interferencia:

Propiedades de seguridad y vida:

Una ***propiedad***de un programa concurrente es un **atributo verdadero en cualquiera de las historias de ejecución** del mismo. Toda propiedad puede ser formulada en términos de dos clases: seguridad y vida.

Son dos aspectos complementarios de la *corrección:*

***Seguridad* (safety): n**ada malo le ocurre a un objeto: asegura estados consistentes

- una *falla de seguridad* indica que algo anda mal

- Ej: ausencia de deadlock y ausencia de interferencia (exclusión mutua) entre procesos.

***Vida* (liveness):** eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks

- una *falla de vida* indica que se deja de ejecutar

- Ej: *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.***

Políticas de scheduling y Fairness:

***Fairness*: trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás.**

Una acción atómica en un proceso es elegible si es la próxima acción atómica en el proceso que será ejecutado

Si hay varios procesos -> hay *varias acciones atómicas elegibles*

**Una *política de scheduling* determina cuál será la próxima en ejecutarse**

**Política: asignar un procesador a un proceso hasta que termina o se demora.**

***Fairness Incondicional***. Una política de scheduling es incondicionalmente fair (o *imparcial*) si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

En el ej. anterior, RR es incondicionalmente fair en monoprocesador, y la ejecución paralela lo es en un multiprocesador (si ningún procesador puede monopolizar el acceso a la vble compartida)

***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

RR es débilmente fair en el ej anterior

No es suficiente para asegurar que cualquier sentencia await elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de false a true y nuevamente a false) mientras un proceso está demorado.

***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.

Requerimientos para los lenguajes de programación.

Los lenguajes de programación deber proveer la capacidad declarar procesos, sincronizar procesos, comunicar procesos, cooperación entre procesos.

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

***De un lenguaje de programación concurrente se requiere:***

• Indicar las tareas o procesos que pueden ejecutarse concurrentemente.

• Mecanismos de sincronización

• Mecanismos de comunicación entre los procesos.

Problemas en sistemas distribuidos.

**3. *Concurrencia con variables compartidas***

**Sincronización por variables compartidas**

Sincronización de grano fino.

Secciones críticas (SC). Definición del problema. Propiedades necesarias de las soluciones:

*Problema de la SC:* implementación de acciones atómicas en software (*locks*)

*Barrera*: punto de sincronización que todos los procesos deben alcanzar para que cualquier proceso pueda continuar.

En la técnica de *busy waiting* un proceso chequea repetidamente una condición hasta que sea verdadera.

Ventaja: implementación con instrucciones de cualquier procesador

Ineficiente en multiprogramación (cuando varios procesos comparten el procesador y la ejecución es interleaved)

Aceptable si cada proceso ejecuta en su procesador.

Propiedades necesarias:

***Exclusión mutua****.* A lo sumo un proceso está en su SC

***Ausencia de Deadlock* (*Livelock*):** Si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.

***Ausencia de Demora Innecesaria:*** Si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.

***Eventual Entrada:*** Un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

Soluciones de tipo spin-locks al problema de la SC.

Solucion spin-locks de grano fino

**bool lock=false; # lock compartido**

**process SC[i=1 to n] {**

**while (true) {**

**while (TS(lock)) skip ; # protocolo de entrada**

**sección crítica;**

**lock = false; # protocolo de salida**

**sección no crítica;**

**}**

**}**

**Solución tipo “spin locks”: los procesos se quedan iterando (*spinning*) mientras esperan que se *limpie* lock**

**Cumple las 4 propiedades si el scheduling es fuertemente fair.**

Una política débilmente fair es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC).

Baja performance en multiprocesadores si varios procesos compiten por el acceso.

*lock* es una vble compartida y su acceso contínuo es muy costoso (“*memory contention*”).

Además, podría producirse un alto overhead por cache inválida

TS escribe siempre en lock aunque el valor no cambie 🡪 tiempo. Mejora 🡪 *Test-and-Test-and-Set*

Spin-locks no controla el orden de los procesos demorados, es posible que alguno non entre nunca si el scheduling no es fuertemente fier.

Algoritmos clásicos de soluciones fair al problema de la SC (tie-breaker, ticket, bakery).

Algoritmo TIE-Breaker: protocolo de Sc que requiere scheduling solo devilmente fair y no usa instrucciones especiales, pero es mas complejo. Usa una variable adicional para comper empares, indicando que proceso fue el ultimo en comenzar a ejecutar su protocolo de entrada a la SC, ultimo es una variable compartida de acceso protegido. Se demora al ultimo en comenzar su protocolo de entrada.

Algotirmo: (n-procesos, complejo y costoso)

int in[1:n] = ([n] 0), ultimo[1:n] = ([n] 0);

process SC[i = 1 to n] {

while (true) {

for [j = 1 to n] { # protocolo de entrada

# el proceso i está en la etapa j y es el último

ultimo[j] = i; in[i] = j;

for [k = 1 to n st i <> k] {

# espera si el proceso k está en una etapa más alta

# y el proceso i fue el último en entrar a la etapa j

while (in[k] >= in[i] and ultimo[j]==i) skip;

}

}

sección crítica;

in[i] = 0;

sección no crítica;

}

}

Algoritmo Ticket: se reparten numeros y se espera su turno. Los clientes toman el numero mayor y esperan su turno.

Cómo se implementa la primera acción atómica??

<turno[i] = numero; numero = numero + 1; >

Sea Fetch-and-Add una instrucción con el siguiente efecto:

FA(var,incr):  <emp = var; var = var + incr; return(temp); >

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;

}

}

En ticket, si no existe FA la solución no puede ser fair.

Algotirmo Bakery: El algoritmo bakery es mas complejo, pero es fair y no requiere instrucciones especiales. No requiere un contador global próximo.

Cada proceso que trata de ingresar recorre los números de los demás y se autoasigna uno mayor. Luego espera a que su número sea el menor de los q esperan. Los procesos se chekean entre ellos y no contra uno global.

int turno[1:n] = ([n] 0);

{BAKERY: (P/todo i: 1<=i <= n: (SC[i] está en su SC) 🡪 (turno[i] > 0) y (p/todo j : 1 <= j <= n, j distinto i: turno[j] = 0 o turno[i] < turno[j] ) ) }

process SC[i = 1 to n] {

while (true) {

< turno[i] = max(turno[1:n] + 1; >

for [j = 1 to n st j <> i]

< await (turno[j] == 0 or turno[i] < turno[j]); >

sección crítica

turno[i] = 0;

sección no crítica } }

Esta solución de grano grueso no es implementable directamente:

- la asignación a turno[i] exige calcular el máximo de n valores

- el await referencia una variable compartida dos veces

Implementación de sentencias Await arbitrarias.

Sincronización barrier. Definición:

Punto de demora al final de cada iteración, es una barrera a la que deben llegar todos antes de permitirles pasar.

Problemas resueltos por algoritmos iterativos que computan sucesivas mejores aproximaciones a una rta, y terminan al encontrarla o al converger. En general manipulan un arreglo, y cada iteración realiza la misma computación sobre todos los elementos del arreglo.

🡪 Múltiples procesos para computar partes disjuntas de la solución en paralelo

En la mayoría de los algoritmos iterativos paralelos cada iteración depende de los resultados de la iteración previa.

Ignorando terminación, y asumiendo *n* tareas paralelas en cada iteración, se tiene la forma general:

while (true) {

co [i=1 to n]

código para implementar la tarea i;

oc

}

Ineficiente, ya que produce *n* procesos en cada iteración.

🡪crear procesos al comienzo y sincronizarlos al final de c/ iteración

**process Worker[i=1 to n] {**

**while (true) {**

**código para implementar la tarea i;**

**esperar a que se completen las *n* tareas; }**

**}**

Soluciones posibles (contador compartido, flags y coordinadores, árboles, barreras simétricas, butterfly):

**Contador Compartido:**

*n* workers necesitan encontrarse en una barrera.

Int cantidad = 0;

process Worker[i=1 to n] {

while (true) {

código para implementar la tarea i;

< cantidad = cantidad + 1; > #*cantidad* incrementado por c/ worker al llegar

< await (cantidad == n); > #cuando *cantidad* es *n*, se les permite pasar

}

} Problema: cantidad necesita ser 0 en cada iteración, puede haber contención d mem, coherencia d cache.

**Flags y Coordinadores:**

Puede “distribuirse” *cantidad* usando *n* variables (arreglo *arribo*[1..n]) El await pasaría a ser  **await (arribo[1] + … + arribo[n] == n);**  Reintroduce memory contention y es ineficiente

Puede usarse un conjunto de valores adicionales y un proceso más. ***Cada Worker espera por un único valor***.

int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);

# *arribo* y *continuar* son “*flags”*

process Worker[i=1 to n] {

while (true) {

código para implementar la tarea i;

arribo[i] = 1;

< await (continuar[i] == 1); >

continuar[i] = 0;

}

}

process Coordinador {

while (true) {

for [i=1 to n] {

< await (arribo[i] == 1); >

arribo[i] = 0;

}

for [i = 1 to n] continuar[i] = 1;

}

}

**Arboles:** Problema: require un proceso extra. Posible solución: combinar acciones de workers y coordinador haciendo que cada worker tamb sea coordinador.

**Barrera Simétrica:** una barrera simétrica se construye a partir de pares simples d barreras para dos procesos.

**Buterfly:** Worker[1:n] arreglo de procesos. Si *n* es potencia de 2  *butterfly barrier*

Algoritmos *data parallel*.. Computación de prefijos:

Algoritmos *data parallel* -> varios procesos ejecutan el mismo código y trabajan en distintas partes de datos compartidos.

**Sincronización por semáforos**

Defectos de la sincronización por variables compartidas:

*Protocolos “busy-waiting”*: complejos y sin clara separación e/ vbles de sincronización y las usadas para computar resultados. Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos. Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso *spinning* puede ser usado de manera más productiva por otro proceso.

🡪 Necesidad de *herramientas* para diseñar protocolos de sincronización.

Semáforos. Sintaxis y semántica.

Semáforo: instancia de un tipo de datos abstracto con solo dos operaciones atómicas P y V.

V: señala la ocurrencia de un evento, incrementa.

P: se usa para demorar un proceso hasta que ocurra un evento, decrementa.

Declaraciones:

sem s;

sem mutex = 1;

sem fork[5] = ([5] 1);

Semáforo general (o *counting semaphore*)

P(s): <await (s > 0) s = s-1; >

V(s): <s = s+1; >

Semáforo binario

P(b): <await (b > 0) b = b-1; >

V(b): <await (b < 1) b = b+1; >

Usos básicos y técnicas de programación:

Permite proteger secciones críticas y pueden usarse para implementar sincronización por condición.

Soluciones a SC y barreras.

*En una barrera p/ 2 procesos, los eventos significativos son las llegadas de los procesos a la barrera* : 2 semáforos de señalización.

sem llega1=0, llega2=0;

process Worker1 {

..........

V(llega1); # señaliza el arribo

P(llega2); # espera al otro proceso

………

}

process Worker2 {

..........

V(llega2); # señaliza el arribo

P(llega1); # espera al otro proceso

………

}

Puede usarse la barrera para dos procesos para implementar una butterfly barrier para n, o sincronización con un coordinador central

Semáforos binarios divididos (*split*):

SBS: Los bi pueden verse como un único semáforo b que fue dividiso en n semáforos. Al lo sumo uno d ellos puede ser 1 a la vez.

Exclusión mutua selectiva:

Un problema es de exclusión mutua selectiva cuando cada proceso compite por el acceso a los recursos, no con todos los demás sino con un subconjunto de ellos.

Ej: Problema de los filósofos, lectores escritores.

La técnica “*passing the baton”*. Definición y aplicaciones:

***Passing the baton*:** cuando un proceso está dentro de una SC mantiene el *baton* (testimonio, token) que significa **permiso para ejecutar**

Cuando el proceso llega a un **SIGNAL**, pasa el *baton* (control) a otro proceso.

Si ningún proceso está esperando una condición que sea ***true,*** el baton se pasa ***al próximo proceso que trata de entrar a su SC*** por primera vez (es decir, uno que ejecuta P(e) ).

Lectores y escritores:

- **e** semáforo p/ controlar acceso a las vbles compartidas, **r** semáforo asociado a la guarda en procesos lectores, y **w** asociado a la guarda en escritores

- **dr** y **dw** contadores de lectores y escritores esperando

Sincronización por condición general.

Alocación de recursos. SJN.

Ejemplos clásicos: filósofos, lectores y escritores, productores y consumidores con buffer limitado, etc.

Semáforos en lenguajes reales: Pthreads. Ejemplos.

**Sincronización por monitores**

Noción de Regiones Críticas Condicionales.

Monitores. Sintaxis y semántica.

Sincronización en monitores.

“Signal and wait“ y “Signal and continue”: diferencias:

***Signal and Continued*:** el proceso que hace el *signal* continúa usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al *wait*).

•***Signal and Wait*:** el proceso que hace el *signal* pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al *wait.*

La técnica “*passing the condition”:*

Ejemplos clásicos: buffer limitado, lectores y escritores, alocación, etc.

Diseño de un reloj lógico. Alternativas.

El problema del peluquero: rendezvous:

***Problema del peluquero dormilón* (*sleeping barber*).**

Una ciudad tiene una peluquería con 2 puertas y unas pocas sillas. Los clientes entran por una puerta y salen por la otra. Como el negocio es chico, a lo sumo un cliente o el peluquero se pueden mover en él a la vez. El peluquero pasa su tiempo atendiendo clientes, uno por vez. Cuando no hay ninguno, el peluquero duerme en su silla. Cuando llega un cliente y encuentra que el peluquero está durmiendo, el cliente lo despierta, se sienta en la silla del peluquero, y duerme mientras el peluquero le corta el pelo. Si el peluquero está ocupado cuando llega un cliente, éste se va a dormir en una de las otras sillas. Después de un corte de pelo, el peluquero abre la puerta de salida para el cliente y la cierra cuando el cliente se va. Si hay clientes esperando, el peluquero despierta a uno y espera que se siente. Sino, se vuelve a dormir hasta que llegue un cliente.

Scheduling de discos. Ejemplo. Enfoques alternativos para la sincronización.

Monitores en lenguajes reales: Java, Pthreads. Ejemplos:

JAVA 🡪 lenguaje OO con soporte para concurrencia, mediante ***Threads***. Un thread es un proceso “liviano” (lightweight process) que tiene un contexto privado mínimo

Concurrencia en Java

Cada Thread en JAVA tiene su propio contador de programa, su pila de ejecución (stack) y su conjunto de registros (working set), pero la zona de datos es compartida por todos los threads, exigiendo sincronización.

**Métodos sincronizados** 🡪 **Monitores**

**Implementaciones**

Conceptos de implementación de procesos en arquitecturas mono y multiprocesador.

Kernel monoprocesador y multiprocesador.

**4. *Programación distribuida. Concurrencia con pasaje de mensajes***

Programas distribuidos:

Programa distribuido: programa concurrente comunicado por mensajes. Supone la ejecución sobre una arquitectura de memoria distribuida, auqnue puedan ejectarse sobre una de memoria compartida o hibrida.

Primitiva de pasaje de mensajes: interfaz con el sistema de comunicaciones. Los procesos comparten canales.

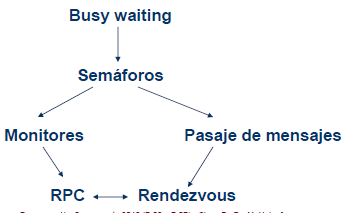
Relación entre mecanismos de comunicación:

Semáforos: mejora respecto de busy waiting;

Monitores: combinan EM implícita y señalización explícita

PM: extiende semáforos con datos;

RPC y rendezvous: combinan la interfase procedural de monitores con PM implícito.



Clases básicas de procesos:

* productores y consumidores:
* clientes
* servidores, peers.

Control de concurrencia en Sistemas Distribuidos.

**Mensajes asincrónicos**

Sintaxis y semántica. Canales. Operaciones.

Filtros. Redes de Filtros.

Clientes/Servidores. Algoritmos clásicos. Monitores activos. Continuidad conversacional.

Peers. Intercambio de valores. Cálculo de la topología de una red.

Mensajes asincrónicos en lenguajes reales MPI. Extensión de lenguajes secuenciales con bibliotecas específicas.

**Mensajes sincrónicos**

Sintaxis y semántica.

Conceptos de CSP.

Comunicación guardada. Sintaxis y semántica:

**Las *sentencias de comunicación guardada* soportan comunicación no determinística:**

**B; C** 🡪 **S;**

B puede omitirse y se asume true.

B y C forman la ***guarda***.

La guarda *tiene éxito* si B es true y ejecutar C no causa demora.

La guarda *falla* si B es falsa. La guarda *se bloquea* si B es true pero C no puede ejecutarse inmediatamente.

Filtros. Clientes y servidores. Asignación de recursos.

Interacción entre procesos paralelos.

Ejemplos.

Mensajes sincrónicos en lenguajes reales: OCCAM. Constructores secuenciales y paralelos. Comunicación y sincronización. Ejemplos

**Linda: extensiones para el manejo de PM**

Linda. Estructuras de datos distribuidas

LINDA: aproximación distintiva al procesamiento concurrente que combina aspectos de MC y PMA.

NO es un lenguaje de programación, sino un conjunto de 6 primitivas que operan sobre una MC donde hay “**tuplas nombradas**” (*tagged tuples*) **que pueden ser *pasivas* (datos) o *activas* (tareas)**. Puede agregarse como biblioteca a un lenguaje secuencial. El núcleo de LINDA es el *espacio de tuplas* compartido (TS) que puede verse como un único canal de comunicaciones compartido, *pero en el que no existe orden*:

- Depositar una tupla (**OUT**) funciona como un **SEND**.

- Extraer una tupla (**IN**) funciona como un **RECEIVE.**

**- RD** permite “leer”como un RECEIVE pero sin extraer la tupla de TS.

**- EVAL** permite creación de procesos (tuplas activas) dentro de TS.

- Por último **INP y RDP** permiten hacer IN y RD no bloqueantes.

Si bien hablamos de MC, TS puede estar físicamente distribuida en una arq. multiprocesador (más coomplejo) 🡪 puede usarse para almacenar estr. de datos distribuidas, y ≠ procesos pueden acceder concurrentemente diferentes elementos de las mismas.

TS: colección no ordenada de tuplas pasivas y activas.

- Tuplas de datos (pasivas): registros tagged que contienen el estado compartido de una computación.

- Tuplas activas (de proceso): rutinas que ejecutan asincrónicamente.

- Cuando una tupla proceso termina se convierte en tupla de datos

Interacción: leyendo, escribiendo y generando tuplas de datos.

Cuando una tupla proceso termina, se convierte en tupla de datos.

Cada tupla de datos en TS tiene la forma: **("tag", value1, ..., valuen)**

“tag” es un string literal para distinguir entre tuplas que representan distintas estructuras de datos.

Las primitivas básicas (*y atómicas*) para manipular tuplas de datos son OUT, IN y RD

El concepto de *bag of tasks*:

Idea: tener una “bolsa” de tareas que pueden ser compartidas por procesos “worker”.

C/ worker ejecuta un código básico

**while (true) {**

***obtener una tarea de la bolsa***

**if (no hay más tareas)**

**BREAK; # exit del WHILE**

***ejecutar tarea* (*incluyendo creación de tareas*)*;***

**}**

Puede usarse p/ resolver problemas con un n° fijo de tareas y p/ soluciones recursivas con nuevas tareas creadas dinámicamente.

***El paradigma de “bag of tasks” es sencillo, escalable (aunque no necesariamente en performance) y favorece el balance de carga entre los procesos.***

Espacio de tuplas e interacción entre procesos.

LINDA 🡪 aproximación distintiva al procesamiento concurrente que combina aspectos de MC y PMA.

NO es un lenguaje de programación, sino un conjunto de 6 primitivas que operan sobre una MC donde hay “**tuplas nombradas**” (*tagged tuples*) **que pueden ser *pasivas* (datos) o *activas* (tareas)**.

Puede agregarse como biblioteca a un lenguaje secuencial.

El núcleo de LINDA es el *espacio de tuplas* compartido (TS) que puede verse como un único canal de comunicaciones compartido, *pero en el que no existe orden*:

- Depositar una tupla (**OUT**) funciona como un **SEND**.

- Extraer una tupla (**IN**) funciona como un **RECEIVE.**

**- RD** permite “leer”como un RECEIVE pero sin extraer la tupla de TS.

**- EVAL** permite creación de procesos (tuplas activas) dentro de TS.

- Por último **INP y RDP** permiten hacer IN y RD no bloqueantes.

Si bien hablamos de MC, TS puede estar físicamente distribuida en una arq. multiprocesador (más coomplejo)  puede usarse para almacenar estructuras de datos distribuidas, y ≠ procesos pueden acceder concurrentemente diferentes elementos de las mismas.

TS 🡪 colección no ordenada de tuplas pasivas y activas.

- Tuplas de datos (pasivas): registros tagged que contienen el estado compartido de una computación.

- Tuplas activas (de proceso): rutinas que ejecutan asincrónicamente.

- Cuando una tupla proceso termina se convierte en tupla de datos

Interacción  leyendo, escribiendo y generando tuplas de datos.

Cuando una tupla proceso termina, se convierte en tupla de datos.

Un proceso deposita una tupla en TS ejecutando:

**out("tag", expr1, ..., exprn)**

- La ejecución de out termina cuando las expresiones fueron evaluadas y la tupla de datos resultante fue depositada en TS.

- Similar a send, pero la tupla se almacena en un TS no ordenado en lugar de ser agregada a un canal específico.

Un proceso extrae una tupla de datos de TS ejecutando:

**in("tag", field1, ..., fieldn)**

- Cada fieldi o es una expresión o un parámetro formal de la forma *?var* donde *var* es una vble local en el proceso ejecutante.

- Los argumentos p/ in son llamados *template*. El proc que ejecuta in se demora hasta que TS contenga al menos 1 tupla que matchee el template, y luego remueve una de TS.

- El *tag* actúa como un nombre de canal.

Ejemplos

**Remote Procedure Calls y Rendezvous.**

Sintaxis y semántica:

RPC y Rendezvous son técnicas de comunicación y sincronización entre procesos que suponen **un canal bidireccional**, ideales para programar aplicación C/S. Combinan una interfaz tipo “monitor” con opraciones exportadas a través de llamadas externas (CALL) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados).

Similitudes y diferencias:

Difieren en la manera de servir la invocación de operaciones.

Un enfoque es declarar un procedure para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada nuevo llamado (RPC porque el llamador y el cuero del procedure pueden estar en distintas maquinas). Para el cliente, durante la ejecución del servicio, es como si tuviera en su sitio el proceso remoto que lo sirve.

El segundo enfoque es hacer rendezvous con un proceso existente. Un rendezbous es servido por un sentencia de entrada o accept que espera un invocación, la procesa y devuelve los resultados.

RPC: Sincronización en módulos.

Discusión revisada de aplicaciones.

RPC en lenguajes reales: Java. RMI. Ejemplos.

Rendezvous en lenguajes reales: Ada. Tasks y sincronización. Ejemplos

**Primitivas múltiples**

La notación de primitivas múltiples.

RPC y rendezvous: un proceso inicia la comunicación con un call, que bloquea al llamador hasta que la operación es servida y se retornan los resultados.

Ideales para interacciones C/S, pero difícil programar algoritmos filtros o peers que intercambian información (para éstos es mejor PMA)

La **Notación de Primitivas Múltiples:** combina **RPC**, **Rendezvous** y **PMA** en un paquete coherente.

- Brinda gran poder expresivo combinando ventajas de las 3 componentes, y poder adicional

- Programas = colecciones de módulos. Una operación visible se declara en la especificación del módulo. Puede ser invocada por procesos de otros módulos, y es servida por un proceso o procedure del módulo que la declara.

- También se usan operaciones *locales*, que son declaradas, invocadas y servidas dentro del cuerpo de un único módulo.

Sintaxis y semántica:

Una operación puede ser invocada por call sincrónico o por send asincrónico:

**call *Mname.op*(argumentos)**

**send *Mname.op*(argumentos)**

El call termina cuando la operación fue servida y los resultados fueron retornados

El send termina tan pronto como los argumentos fueron evaluados

Una operación es servida por un procedure (proc) o por rendezvous (sentencias in). La elección la toma el programador del módulo.

MPD. Componentes de programa. Comunicación y sincronización. Ejemplos

**Implementaciones**

Conceptos de implementación de mecanismos de comunicación y sincronización en ambientes distribuidos.

**5. *Overview de lenguajes de programación***

Resolución de problemas mediante diferentes paradigmas de interacción entre procesos:

* Servidores replicados: los procesos manejan recursos compartidos tales como dispositivos o archivos.
* algoritmos heartbeat: Los procesos periódicamente deben intercambiar información con mecanismos tipo send/recibe.
* algoritmos pipeline: la información recorre una serie de procesos utilizando alguna forma de receive/send.
* prueba-eco: la interaccion entre los procesos permiten recorrer graficos o arboles diseminando y juntando informacion.
* Broadcast: permiten alcanzar una informacion global en una arquitectura distribuida. Sirven para tomar deciciones descentralizadas
* token passing: en muchos casos la arquitectura distribuida recibe una información global a travez del viaje de tokens de control o datos. Tambiénpermite la toma de deciciones distrbuidas.
* manager/workers: implementación distribuida del modelo de bag of tasks.

Ejemplos. Comparación de alternativas.

**6. *Introducción a la Programación Paralela***

El objetivo programación paralela es reducir el tiempo de ejecución, o resolver problemas más grandes o con mayor precisión en el mismo tiempo.

Computación científica:

Los dos modos tradicionales del descubrimiento científico son *teoría* y *experimentación*.

El 3er modo es la *modelización computacional*, que usa computadoras para simular fenómenos y tratar cuestiones del tipo “*what if?*”

Entre las diferentes aplicaciones de cómputo científicas y modelos computacionales existen tres técnicas fundamentales:

(1) Computación de grillas (soluciones numéricas a PDE, imágenes). Dividen una región espacial en un conjunto de puntos.

(2) Computación de partículas (modelos que simulan interacciones de partículas individuales como moléculas u objetos estelares)

(3) Computación de matrices (sistemas de ecuaciones simultáneas)

Necesidad del paralelismo:

Ej: *simulación de circulación oceánica*. División del océano en 4096x1024 regiones, y c/u en 12 niveles. Aprox. 50 millones de celdas 3D. Una iteración del modelo simula la circulación por 10 minutos y requiere alrededor de 30 billones de cálculos de punto flotante. Se intenta usar el modelo para simular la circulación en un período de años...

Problemas “grand challenge”. Abarcan química cuántica, mecánica estadística, cosmología y astrofísica, dinámica de fluidos computacional, diseño de materiales, biología, farmacología, secuencia genómica, ingeniería genética, medicina y modelización de órganos y huesos humanos, pronóstico del tiempo, sensado remoto, física de partículas etc.

Diseño de algoritmos paralelos:

**No se reduce a simples recetas, sino que es necesaria la *creatividad.***

**La mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes**

**Puede darse un enfoque metódico p/ maximizar el rango de opciones, brindar mecanismos para evaluar las alternativas, y reducir el costo de *backtracking* por malas elecciones**

🡪 Metodología de diseño que da un enfoque exploratorio en el cual aspectos independientes de la máquina tales como la concurrencia son considerados temprano, y los aspectos específicos de la máquina se demoran.

4 etapas: - *particionamiento* (descomposición en tareas) - *comunicación* (estructura necesaria para coordinar la ejecución) - *aglomeración* (evaluación de tareas y estructura con respecto a performance y costo, combinando tareas para mejorar) - *mapeo* (asignación de tareas a procesadores)

Métricas. Conceptos de speedup y eficiencia:

**Speed UP:** Medida de la mejora de rendimiento (performance) de una aplicación al aumentar la cantidad de procesadores (comparando con el rendimiento al utilizar un solo procesador).

Speed UP= tiempo secuencial -🡪 mejor tiempo sec.

Tiempo paralelo 🡪 peor tiempo paralelo

**Eficiencia:** es una medida relativa que permite la comparación de desempeño en diferentes entornos de computación paralela.

La ley de Amdahl:

La ley de Amdahl indica que para un problema dado, existe un speed up máximo alcanzable independientemente del número de procesadores. Significa que es el algoritmo el que decide la mejora de la velocidad dependiendo de la cantidad de código no paralelizadle y no del número de procesadores, llegando a un punto en que no se puede paralelizar el algoritmo.

Concepto de escalabilidad: