**Conceptos Básicos**

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 conectando a los otros a través de una red.

Proceso: programa secuencial.

Un único thread de control -> programación secuencial, monoprocesador.

Múltiples threads de control -> procesos paralelos, programa concurrente.

Los procesos se comunican (memoria compartida o pasaje de mensajes).

Los procesos se sincronizan, exclusión mutua en el acceso a secciones críticas de código para no ejecutar simultáneamente, y sincronizan por condición.

Programa concurrente: 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.

Concurrencia: concepto de software no restringido a una arquitectura en particular de hardware ni a un número 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 procesamiento de datos, a través de un enfoque diferente de la arquitectura lógica de las soluciones.

Algunas ventajas son la velocidad de ejecución que se puede alcanzar, la mejor utilización de la CPU de cada procesador, y la explotación de la concurrencia inherente a la mayoría de los problemas reales.

**Áreas de estudio en la Programación Concurrente**

Elección de granularidad: para una aplicación, significa optimizar la relación entre el número de procesadores y el tamaño de memoria local.

Manejo de recursos: asignación de recursos compartidos, métodos de accesos a los recursos, seguridad y consistencia de los recursos.

Sincronización: se debe asegurar un orden temporal entre las acciones que ejecutan los procesos. Las tareas se intercalan en el tiempo, deben fijarse restricciones. El objetivo de la sincronización es restringir las historias de un programa concurrente sólo a las permitidas.

**Conceptos de Hardware**

Multiprocesadores con memoria distribuida: procesadores conectados por una red. Cada uno tiene memoria local y la interacción es solo por pasaje de mensajes.

Multiprocesadores de memoria compartida: la interacción se da modificando datos almacenados en la Memoria Compartida.

**Problemas asociados con la Programación Concurrente**

La mayor complejidad es que los lenguajes de programación tienen requerimientos adicionales. Aumenta el tiempo de desarrollo y puesta a punto respecto de los programas secuenciales, y puede aumentar el costo de los errores -> mayor costo de los ambientes y herramientas de IS de Sistemas Concurrentes.

Para obtener una real mejora de performance, se requiere adaptar el software concurrente al hardware paralelo (mapeo).

Los procesos no son independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas -> menos confiabilidad.

Los procesos iniciados dentro de un programa concurrente pueden no estar “vivos”. Esta pérdida de la propiedad de liveness puede indicar deadlocks o una mala distribución de recursos.

Hay un no determinismo implícito en el interleaving de procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas -> dificultad para la interpretación y el debug.

**Mecanismos de comunicación y sincronización entre procesos**

**Memoria Compartida**: los procesos intercambian mensajes sobre la memoria compartida o actúan coordinadamente sobre los datos residentes en ella.

Lógicamente no pueden operar simultáneamente sobre la MC, 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 MC.

**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 transmitir mensajes.

Independientemente del mecanismo de comunicación / sincronización entre los 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.

**Paradigma de Resolución de Programas Concurrentes**

Los patrones de resolución concurrentes son pocos:

1. Paralelismo Iterativo: un programa consta de un conjunto de procesos (posiblemente idénticos) cada uno de los cuales tiene uno o más loops. Luego cada proceso es un programa iterativo. Los procesos cooperan para resolver un único problema, pueden trabajar independientemente, y comunicarse y sincronizar por memoria compartida o MP.
2. Paralelismo Recursivo: el problema general puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos.
3. Productores y Consumidores: 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.
4. Clientes y servidores: esquema dominante en las aplicaciones de procesamiento distribuido. Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Unos y otros puede ejecutarse en procesadores diferentes. Los mecanismos de invocación son variados (rendezvous y rpc por ejemplo). El soporte distribuido puede ser muy simple (LAN) o extendido a toda la web.
5. Pares que interactúan: los procesos (que forman parte de un programa distribuido) resuelven parte del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea. El esquema permite mayor grado de asincronismo que C/S.

**Volviendo al Hardware…**

Podemos identificar diferentes enfoques para clasificar las arquitecturas paralelas:

- *Por la organización del espacio de direcciones* (memoria compartida/memoria distribuida).

- *Por el mecanismo de control*: se centra en la manera en que las instrucciones son ejecutadas sobre los datos.

SISD: single instruction single data. Instrucciones ejecutadas en secuencia, una por ciclo de instrucción. La memoria afectada es usada sólo por esta instrucción. Usada por la mayoría de los uniprocesadores. La CPU ejecuta las instrucciones (decodificadas por la UC) sobre los datos. La memoria recibe y almacena los datos en las escrituras, y brinda datos en las lecturas.

MISD: multiple instruction single data. Los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes. Operación sincrónica. No son máquinas de propósito general.

SIMD: single instruction multiple data. Un conjunto de procesadores idénticos con sus propias memorias, que ejecutan la instrucción sobre distintos datos. El host hace broadcast de la instrucción es asincrónica. Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones.

MIMD: multiple instruction multiple data. Cada procesador tiene su propio flujo de instrucciones y de datos -> c/u ejecuta su propio programa. Pueden ser con memoria compartida o distribuida.

- *Por la granularidad*.

De grano grueso: pocos procesadores muy poderosos.

De grano fino: gran número de procesadores menos potentes.

De grano medio.

Las diferentes aplicaciones son adecuadas para una u otra clase a distintos grados:

- si tienen concurrencia limitada pueden usar eficientemente pocos procesadores -> conviene maquinas de grano grueso.

- las máquinas de grano fino son más efectivas en costo para aplicaciones con alta concurrencia.

- *Por la red de interconexión*: tanto las máquinas de MC como de MP pueden construirse conectando unidades de procesador y memoria usando diversas redes de interconexión, estáticas y dinámicas.

Las **redes estáticas** constan de links punto a punto. Se usan para máquinas de MP.

Las **redes dinámicas** están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de MC.

**Acciones Atómicas y Sincronización. Atomicidad de Grano Fino.**

***Acción atómica***: hace una transformación de estado indivisible. Los estados intermedios en la implementación de la acción no deben ser visibles por los otros procesos. Una acción atómica de grano fino 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.

Pero normalmente, los programas concurrentes no son disjuntos –> es necesario establecer algún requerimiento más débil.

**Propiedad de “A lo sumo una vez”**.

Una ***referencia crítica*** en una expresión es una 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, ó
2. **e**  no contiene referencias críticas, en cuyo caso **x**  puede ser leída por otro proceso.

**Especificación de la Sincronización**

**<e>** indica que la expresión **e** debe ser evaluada atómicamente.

<await (B), S;> se utiliza para especificar sincronización.

La expresión booleana B especifica una condición de demora. S es una secuencia de sentencias que se garantiza que termina. Se garantiza que B es True cuando comienza la ejecución de S. Ningún estado interno de S es visible para los otros procesos.

**<S>** solo exclusión mutua.

**<await (B) >** solo sincronización por condición.

Si B satisface ASV, puede implementarse como busy waiting o spinning: do (not B) -> skip od

**Semáforo**

Semáforo: Instancia de un tipo de datos abstracto (o un objeto) con sólo 2 operaciones (métodos) atómicas: P y V.

Analogía con la sincronización del tránsito para evitar colisiones.

El valor de un semáforo es no negativo.

V señala la ocurrencia de un evento (incrementa).

P se usa para demorar un proceso hasta que ocurra un evento (decrementa).

Permiten proteger Secciones Críticas y pueden usarse para implementar sincronización por condición

Declaraciones:

sem s;

sem mutex = 1;

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

Semáforo general

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; >

Si la implementación de la demora por operaciones P se produce sobre una cola, las operaciones son fair.

Semáforos. Barreras: usar un semáforo para cada flag de sincronización. Un proceso setea el flag ejecutando V, y espera a que un flag sea seteado y luego lo limpia ejecutando P.

*Barrera para dos procesos*. Necesitamos saber cada vez que un proceso llega o parte de la barrera -> hay que relacionar los estados de los dos procesos.

*Semáforo de señalización ->* usualmente inicializado en 0. Un proceso señala un evento ejecutando V(s); otros procesos esperan la ocurrencia de un evento ejecutando P(s).

**Pthreads**

Un ***thread*** es un 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).

Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones “multithreading”.

En principio estos mecanismos fueron heterogéneos y poco portables 􀃎 a mediados de los 90 la organización POSIX auspició el desarrollo de una biblioteca en C para multithreading ***(Pthreads)*** La biblioteca está disponible actualmente en varias versiones de Unix.

*Con esta biblioteca se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.*

Los threads pueden sincronizar por semáforos (semaphore.h es el archivo que contiene las definiciones y operaciones para los semáforos con Pthreads).

Los semáforos pueden declararse globales a threads que los van a utilizar: ***sem\_t mutex;***

En la operación sem\_init se inicializa un semáforo, indicando por ejemplo si el semáforo es compartido por threads de diferentes procesos, o sólo por los threads del mismo proceso:

***sem\_init (&mutex, SHARED, 1);***

Las operaciones ***sem\_wait(&mutex)*** y ***sem\_post(&mutex)*** equivalen a P y V sobre el semáforo.

**Regiones Críticas Condicionales (CCRs)**

* EM implícita y SxC explícita.
* Generalmente las CCRs son más simples de usar que los semáforos

La implementación de *region* es más cara pues las condiciones de demora deben reevaluarse al cambiar una variable compartida. Son el paso histórico de semáforos a monitores.

Introdujeron el uso de condiciones de sincronización booleanas, lo cual se usó luego en varias notaciones de lenguajes.

CCRs imponen restricciones al compilador en el uso de variables compartidas -> programas más estructurados y sistema de prueba más simple pues la interferencia se evita automáticamente.

*Declaración de Resource y sentencias Region*

resource *r* (declaraciones de variables) Es una colección nombrada de variables compartidas a la cual se

requiere acceso mutuamente exclusivo.

Cada variable compartida debe pertenecer a un recurso.

Las variables de un recurso pueden ser accedidas *sólo* dentro de sentencias region que nombran el recurso region *r* [ when *B* ] → *S* end

*-B* y *S* pueden referenciar variables de *r* y variables locales al proceso.

-Demora al proceso hasta que *B* es true; luego se ejecuta *S.*

-La ejecución de region que nombran el mismo recurso es mutuamente exclusiva -> *B* se garantiza que es true cuando comienza la ejecución de *S.*

-Dos region pueden ser anidadas si nombran distintos recursos.

-Un compilador para un lenguaje con CCRs puede chequear que las variables compartidas estén en recursos y sean accedidas sólo dentro de región.

**Monitores**

Semáforos:

-Variables compartidas globales a los procesos.

-Sentencias de control de acceso a la SC dispersas en el código.

-Al agregar procesos, se debe verificar que las VC se accedan correctamente.

-Aunque EM y SxC son conceptos ≠, se programan de forma similar.

**Monitores:** módulos de programa que brindan más estructura y pueden ser implementados eficientemente como los semáforos.

Mecanismo de abstracción de datos:

• encapsulan las representaciones de objetos (recursos).

• proveen un conjunto de operaciones que son *los únicos* medios para manipular la representación.

Contiene variables que almacenan el estado del recurso y procedimientos que implementan las operaciones sobre él.-

*EM ->* implícita asegurando que los procedures en el mismo monitor no ejecutan concurrentemente.

*SxC ->* explícita con variables condición.

Programa Concurrente -> procesos activos y monitores pasivos Dos procesos interactúan invocando procedures en el mismo monitor.

Ventajas:

- un proceso que invoca un procedure puede ignorar cómo está implementado.

- el programador del monitor puede ignorar cómo o dónde se usan los procedures del mismo -> *pueden diseñarse cada proceso y el monitor en forma relativamente independiente.*

Un monitor agrupa la representación y la implementación de un recurso compartido (clase). Tiene interfase y cuerpo.

La interfase especifica las operaciones (métodos) que brinda el recurso.

El cuerpo tiene variables que representan el estado del recurso y procedures que implementan las operaciones de la interfase.

Notación

monitor NombreMonitor {

declaraciones de variables permanentes;

código de inicialización

procedure op1 (par. formales1) {

cuerpo de op1 }

.......

procedure opn (par. formales) {

cuerpo de opn }

}

Lo que distingue a un monitor de un TAD en procesos secuenciales, es que es compartido por procesos que ejecutan concurrentemente.

1. Sólo los nombres de los procedures son visibles desde afuera. Sintácticamente, los llamados al monitor tienen la forma:

call NombreMonitor.opi (argumentos)

2. Los procedures pueden acceder sólo variables permanentes, sus variables locales, y parámetros que le sean pasados en la invocación.

3. El programador de un monitor no puede conocer a priori el orden de llamado de los procedures ⇒ se define un invariante del monitor que especifica los estados “razonables” de las variables permanentes cuando ningún proceso las está accediendo.

La exclusión mutua en monitores se provee implícitamente.

La sincronización por condición es programada explícitamente usando variables condición → cond cv;

Sincronización

El valor asociado a cv es una cola de procesos demorados, no visible directamente al programador.

empty(cv)→ retorna true si la cola controlada por cv está vacía.

wait(cv)→ el proceso se demora al final de la cola de cv y deja el

acceso exclusivo al monitor.

signal(cv)→ despierta al proceso que está al frente de la cola y lo

saca de ella (cola vacía= skip). Ese proceso puede ejecutar cuando

readquiera el acceso exclusivo al monitor.

El proceso que ejecuta SIGNAL retiene el control exclusivo del monitor y puede seguir ejecutando (signal-and-continue)

WAIT y SIGNAL son similares a P y V, pero hay diferencias:

* wait siempre demora un proceso hasta un signal posterior.
* signal no tiene efecto si ningún proceso está demorado sobre la variable condición.

El proceso que hace SIGNAL siempre ejecuta antes (retiene el control) que un proceso despertado como resultado del mismo SIGNAL (S&C).

S&C: usada dentro de Unix, Java y Pthreads

Operaciones adicionales

wait(cv, rank) → wait con prioridad: permite tener más control sobre el orden en que los procesos son encolados y despertados- Los procesos demorados en cv son despertados en orden ascendente de rank.

minrank(cv) → función que retorna el mínimo ranking de demora-

signal\_all (cv) → despierta todos los procesos demorados en cv. El tiempo en que cada uno reinicie efectivamente la ejecución dependerá de las condiciones de exclusión mutua.

Con semántica S&C, es lo mismo que while (not empty(cv)) signal(cv);

**Programación Distribuida**

Los métodos de sincronización que hemos visto se basan en leer y escribir ***variables compartidas***, lo cual significa que los programas concurrentes se ejecuten sobre hardware con ***acceso a memoria compartida.***

Las arq. de memoria distribuida son cada vez más habituales.

Esto implica *procesadores + memoria local + red de comunicaciones*

***+ otro mecanismo de comunicación/sincronización***

-> ***intercambio de mensajes***

Para escribir programas sobre una arquitectura de memoria distribuida, es necesario ***definir la interfaz con el sistema de comunicaciones***

-> ***primitivas de pasaje de mensajes* (“semáforos + datos + sincronización”)**

-> ***los procesos comparten canales*** (físicos o lógicos).

Los programas concurrentes que se comunican por mensajes se denominan ***programas distribuidos.*** Esto supone la ejecución sobre una arquitectura de memoria distribuida, aunque puedan ejecutarse sobre una arquitectura de memoria compartida (o híbrida). En un programa distribuido los canales son el único objeto que los procesos comparten.

Las variables son locales a un proceso (su “*cuidador”*)

-> No hay acceso concurrente a variables y la exclusión mutua no requiere ningún mecanismo especial.

-> Los procesos interactúan comunicándose

**El tema principal es sincronizar la comunicación interproceso**

La manera de hacerlo depende del patrón de interacción entre procesos: productores y consumidores, clientes y servidores, o pares interactuantes.

Diferentes mecanismos para la programación distribuida, que varían en la forma de nombrar y usar los canales, y la manera en que se sincroniza la comunicación.

Los canales pueden brindar comunicación half-duplex o full-duplex.

A su vez estas comunicaciones pueden ser asincrónicas (no bloqueantes) o sincrónicas (bloqueantes).

Las combinaciones dan lugar a AMP, SMP, RPC y Rendezvous que son los 4 mecanismos (equivalentes funcionalmente ya que pueden intercambiarse) cuya sintaxis y semántica serán nuestro objetivo.

Si bien decimos que son equivalentes, veremos que AMP y SMP son mejores para modelos tipo productor-consumidor o pares interactivos, mientras RPC y Rendezvous serán más útiles para esquemas Cliente-Servidor.

Los semáforos son una mejora respecto de busy waiting; los monitores combinan exclusión implícita con señalización explícita; el pasaje de mensajes extiende semáforos con datos; RPC y rendezvous combinan la interfase procedural de monitores con pasaje de mensajes implícito.

**Pasaje de Mensajes Asincrónicos**

En PMA los canales son colas de mensajes que han sido enviados y aún no recibidos:

chan *ch*(*id*1 : *tipo*1, ... , *id*n : *tipo*n )

chan input(char);

chan disk\_access(INT cylinder, INT block, INT count, CHAR\* buffer);

chan result[n] (INT); {arreglo de canales, de 0 a n-1}

Un proceso agrega un mensaje al final de la cola (“ilimitada”) de un canal ejecutando un send, que no bloquea al emisor

send *ch*(*expr*1, ... , *expr*n);

Un proceso recibe un mensaje desde un canal con un receive, que demora al receptor hasta que en el canal haya al menos un mensaje; luego toma el primer mensaje y lo almacena en variables locales

receive ch(var1, ... , varn);

Las variables del receive deben tener los mismos tipos que la declaración del canal ch.

Receive es una primitiva bloqueante, ya que produce un delay.

Sin embargo la semántica es que el proceso NO hace nada hasta recibir un mensaje en la cola correspondiente al canal ch; NO es necesario hacer polling (aquí tendríamos busy waiting).

El acceso a los contenidos de cada canal es atómico y se respeta el orden FIFO.

En principio los canales son ilimitados, aunque las implementaciones reales tendrán un tamaño de buffer asignado.

Se supone que los mensajes NO se pierden ni modifican y que todo mensaje enviado en algún momento puede ser “leído”. empty(*ch*) determina si la cola de un canal está vacía Útil cuando el proceso puede hacer trabajo productivo mientras espera un mensaje

Es una primitiva que debe usarse con cuidado.

La evaluación de empty podría ser true, y sin embargo existir un mensaje al momento de que el proceso reanuda la ejecución.

O podría ser false, y no haber más mensajes cuando sigue ejecutando (si no es el único en recibir por ese canal).

**Pasaje de Mensajes Sincrónicos**

La primitiva de trasmisión sync\_send es bloqueante.

-> El trasmisor queda esperando que el mensaje sea recibido.

*La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje* ⇒ *MENOS memoria.*

Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (*siempre un proceso se bloquea*).

Si bien send y sync\_send son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de deadlock mayores en comunicación sincrónica.