**Clase 1**

¿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 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 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.

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.

Concurrencia a nivel hardware:

**Multiprocesadores de memoria compartida:** La interacción se da modificando datos almacenados en la MC. Problema de consistencia

* Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problema de sincronización y *consistencia*
* Esquemas NUMA para mayor número de procesadores distribuidos.

**Multiprocesadores con memoria distribuida.**

Procesadores conectados por una red. Cada uno 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.

***P*roceso:** cuando tiene un único *thread* o flujo de control: programación secuencial, monoprocesador

Si tiene Múltiples *threads* o flujos de control hablamos de programas concurrentes, procesos paralelos.

**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 **(en paralelo)**, 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.

**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) estos son 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). 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.

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

**Programa secuencial**: Totalmente ordenado, deterministico (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.

**Hay 3 grandes clases (superpuestas) de aplicaciones concurrentes**:

- Sistemas multithreaded: Ejecución de N procesos independientes en M procesadores (N>M). Maneja simultáneamente tareas independientes, asignando los procesadores de acuerdo a alguna política (ej, por tiempos).

- Sistemas de 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.

- Sistemas de cómputo paralelo: Resolver un problema en el menor tiempo (o un problema + grande en aprox. el mismo tiempo) usando una arquitectura multiprocesador en la que se pueda distribuir la tarea global en tareas que puedan ejecutarse en distintos procesadores.

**El procesamiento paralelo lleva a los 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 secuencial.

Tiempo paralelo 🡪 peor tiempo paralelo

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

Los procesos cooperan (Los procesos se combinan para resolver una tarea común , para eso necesitas sincronización) y compiten (Típico en SO y redes, debido a recursos compartidos). 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:

**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 (ej: semáforos).

**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.

**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.

Prioridad, 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.

**Administracion de recursos compartidos:** 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 más 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 está esperando adquirir.

**Clase 2**

Paradigma de resolución de programas concurrentes: Si bien el número de aplicaciones es muy grande, en general los “patrones” de resolución concurrentes son pocos:

* **Paralelismo iterativo**: En el paralelismo iterativo un programa consta de un conjunto de procesos (posiblemente idénticos) c/u de los cuales tiene 1 o más loops 🡪 cada proceso es un programa iterativo. Los procesos cooperan para resolver un único problema (x ej un
* sistema de ecuaciones), pueden trabajar independientemente, y comunicarse y sincronizar por memoria compartida o MP. Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones.
* **Paralelismo recursivo**: 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)
* **Productores y consumidores (pipelines o workflows)**: 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.
* **Clientes y servidores**: 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.
* **Pares que interactúan (interacting peers)**: 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

**Clasificacion de la arquitecturas paralelas**

* Por la organización del espacio de direcciones:
  + Multiprocesadores de memoria compartida: Interacción modificando datos en la memoria compartida. Problema de consistencia
  + Multiprocesadores con memoria distribuida: Memoria local (no hay problemas de consistencia). Interacción es sólo por pasaje de mensajes.
* Por el mecanismo de control: Se basa en la manera en que las instrucciones son ejecutadas sobre los datos.
  + SISD (Single Instruction Single Data): Instrucciones ejecutadas en secuencia, una x ciclo de instrucción. La memoria afectada es usada sólo por esta instrucción. Usada por la mayoría de los uniprocesadores. La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memo recibe y almacena datos en las escrituras, y brinda datos en las lecturas. Ejecución determinística.
  + SIMD (Single Instruction Multiple Data): Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos. Los procesadores en general son muy simples. El host hace broadcast de la instr. Ejecución sincrónica y determinística. Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones. Adecuados para aplicaciones con alto grado de regularidad, (x ej. procesamiento de imágenes).
  + 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
  + MIMD (Multiple Instruction Multiple Data): C/ procesador tiene su propio flujo de instrucciones y de datos, c/u ejecuta su propio programa. Pueden ser de memoria compartida o distribuida.
* Por la granularidad: Relación entre el número de procesadores y el tamaño de memoria total. Puede verse también como la relación entre cómputo y comunicación.
  + 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).

Si tienen concurrencia limitada pueden usar eficientemente pocos procesadores convienen máquinas 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 procesadores y memorias usando diversas redes de interconexión, estáticas y dinámicas. Las redes estáticas constan de links punto a punto. Típicamente se usan para máquinas de MP. Las redes dinámicas están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de MC

**Acciones atomicas, sincronización y conceptos**

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).

La ejecución de un programa concurrente se realiza mediante un intercalado (interleaving) de las acciones atómicas ejecutadas por procesos individuales.

Interacción 🡪 no todos los interleavings son aceptables.

A la Historia de un programa concurrente se la denomina **trace**

Para **sicronizar** se pueden combinar acciones atómicas de grano fino (fine-grained) en acciones (compuestas) de grano grueso (coarse grained) que den la exclusión mutua, o también demorar un proceso hasta que el estado de programa satisfaga algún predicado (por condición). El objetivo de la sincronización es prevenir los interleavings indeseables restringiendo las historias de un programa concurrente sólo a las permitidas

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

**CLASE 3**

**Acciones atómicas y Sincronización. Atomicidad de grano fino**

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 tambien será atómica.

La **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 (ASV) 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

Esta propiedad asegura atomicidad ya que la variable compartida, si la hay, será leída o escrita solo una vez como una acción atómica de grado fino.

Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente. Una acción atómica de grano grueso es una especificación en alto nivel del comportamiento requerido de un programa que puede ser implementada de distintas maneras, dependiendo del mecanismo de sincronización disponible.

**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.

**Sincronización por variables compartidas. Locks y barreras**

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, la ventaja esta en la implementación, que se realiza con instrucciones de cualquier procesador, pero es ineficiente en multiprogramación (cuando varios procesos comparten el procesador y la ejecución es interleaved). Es 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.

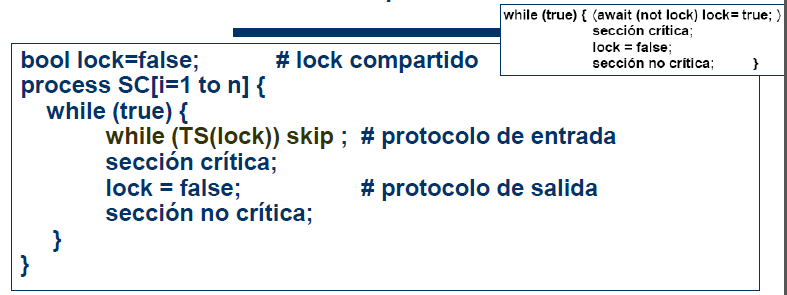
**Ausencia de Demora Innecesaria:** Si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.

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

*Las 3 primeras son propiedades de seguridad, y la 4° de vida*

La solución trivial es hacer atómica la sección critica, pero como se implementa?

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



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

Además, podría producirse un alto overhead por cache inválida. Ademas TS escribe siempre en lock aunque el valor no cambie lo que afecta al tiempo 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 fair.

Spin locks no controla el orden de los procesos demorados, es posible que alguno no entre nunca si el scheduling no es fuertemente fair (race conditions)

**Algoritmo Tie-Breaker (o algoritmo de Peterson):** protocolo de SC que requiere scheduling sólo débilmente fair y no usa instrucciones especiales, pero es más complejo

Usa una variable adicional para romper empates, indicando qué proceso fue el último en comenzar a ejecutar su protocolo de entrada a la SC, esa variable llamado ultimo es compartida de acceso protegido. Se demora (quita prioridad) al último en comenzar su protocolo de entrada.

Tie-Breaker para muchos procesos (nprocesos) es costoso en tiempo y complejo.

**Algoritmo Ticket:** se reparten números y se espera turno. Los clientes toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los clientes con un número más chico sean atendidos.

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

**Bakery** es más complejo, pero es fair y no requiere instrucciones especiales. No requiere un contador global proximo que se “entrega” a cada proceso al llegar a la SC. 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 que esperan. Los procesos se chequean entre ellos y no contra un global

**Sincronizacion Barrier**: 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.

* **Contador compartido**: La manera más simple de especificar los requerimientos para una barrera es emplear un entero compartido, count, el cual es inicialmente 0. Asumimos que hay n procesos worker que necesitan encontrarse en una barrera. Cuando un proceso llega a la barrera, incrementa count. Por lo tanto, cuando count es n, todos los procesos pueden continuar.   
  La dificultad es que count debe ser 0 al comienzo de cada iteración. Por lo tanto, count necesita ser reseteada a 0 cada vez que todos los procesos han pasado la barrera. Más aún, tiene que ser reseteada antes de que cualquier proceso trate nuevamente de incrementar count. Es posible resolver este problema de “reset” empleando dos contadores, uno que cuenta hasta n y otro que cuenta hacia abajo hasta 0, con los roles de los contadores switcheados después de cada etapa 🡪 Esto podría llevar a memory contention
* **Flags y coordinadores:** Una manera de evitar el problema de contención de memoria es distribuir la implementación de count usando n variables que sumen el mismo valor. Podemos resolver los problemas de contención y reset usando un conjunto adicional de valores compartidos y empleando un proceso adicional, Coordinator. En lugar de que cada Worker tenga que sumar y testear los valores de arrive, hacemos que cada Worker espere que un único valor se convierta en true. Hay un proceso coordinador que espera a que todos los elementos de arrive se vuelvan 1, luego setea todos los elementos de continue en 1.
* **Barreras simetricas:** Una barrera simétrica para n procesos se construye a partir de pares de barreras simples para dos procesos. Cómo se combinan para construir una barrera n proceso??? 🡪 Worker[1:n] arreglo de procesos. Si n es potencia de 2 🡪 butterfly barrier. log2n etapas: cada Worker sincroniza con uno distinto en c/ etapa. En la etapa s, un Worker sincroniza con otro a distancia 2s-1. Cuando cada Worker pasó log2n etapas, todos deben haber llegado a la barrera y por lo tanto todos pueden seguir.

**Computacion de prefijo paralelo**

**Algoritmos data paralle🡪** varios procesos ejecutan el mismo código y trabajan en distintas partes de datos compartidos.

Ejemplo: computar en paralelo las sumas de los prefijos de un arreglo a[n], para obtener sum[n], donde sum[i] es la suma de los primeros i elementos de a. ¿Cómo se puede paralelilzar? 🡪 Idea: Sumar en paralelo pares contiguos, luego pares a distancia 2, luego a distancia 4, etc. = (log2n) pasos.

**Defectos de la sincronización busy waiting**: 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.

**Clase 4**

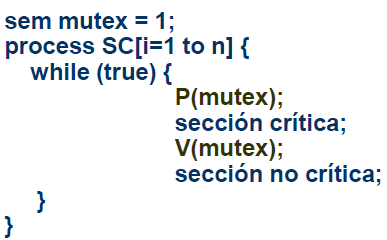
**Semaforos:** instancia de un tipo de datos abstracto (o un objeto) con sólo 2 operaciones. (métodos) atómicas: P y V

P(s): 〈 await s > 0 → s := s - 1 〉

V(s): 〈 s := s + 1 〉

**Problema de la sección critica:**

**Exclusion mutua:** Con un semáforo inicilaizado en uno podemos solucionar el problema de acceso a las secciones criticas.

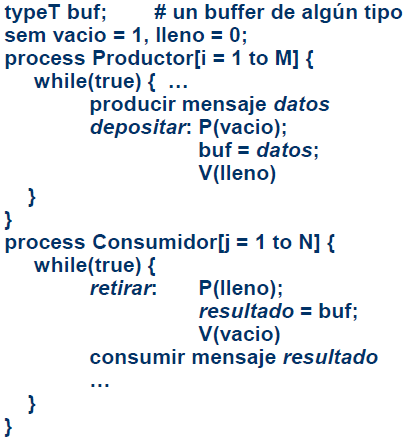


**Barreras:** un semáforo para cada flag de sincronización. Un proceso setea el flag ejecutando V, y espera a que un flag sea seteado y luego lo limpia ejecutando P

**Barrera para dos procesos:** Necesitamos saber cada vez que un proceso llega o parte de la barrera 🡪 relacionar los estados de los dos procesos

**Semáforo de señalización:**  generalmente inicializado en 0.Un proceso señala el evento con V(s); otros procesos esperan la ocurrencia del evento ejecutando P(s).

**Productores y múltiples consumidores**: Dos operaciones: depositar y retirar deben alternarse. Dos semáforos (vacío y lleno).Inicialización: vacío en 1 y lleno en 0.

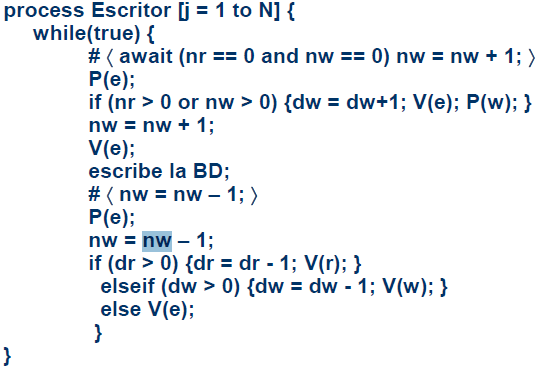
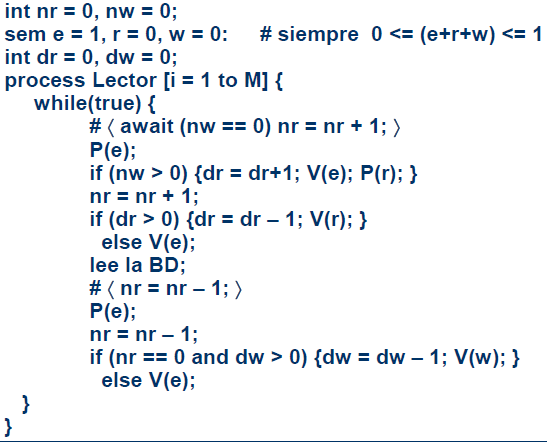


**EXCLUSION MUTUA SELECTIVA**

**Problema de los filosofos:** si todos los filósofos levantaron un tenedor, luego ninguno puede levantar su segundo. Una condición necesaria para deadlock es que hay waiting circular, es decir, un proceso está esperando un recurso tomado por un segundo, el cual espera un recurso tomado por un tercero, y así siguiendo hasta un proceso que espera un recurso tomado por el primero. Así, para evitar deadlock es suficiente asegurar que no puede ocurrir waiting circular. Para este problema, una aproximación es tener uno de los procesos, digamos Philosopher[5], levantando sus tenedores en orden inverso.

**Lectores y Escritores:** El acceso de los escritores debe ser exclusivo para evitar interferencia entre transacciones. Los lectores pueden ejecutar concurrentemente entre ellos si no hay escritores actualizando.

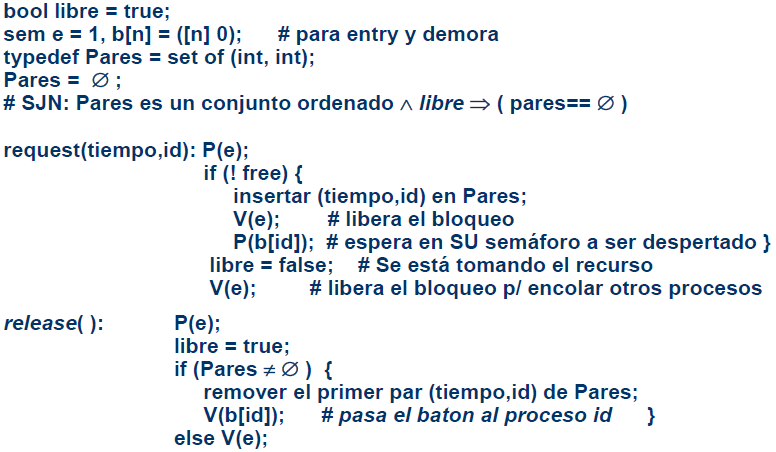
**Passing the baton**



**Alocación de Recursos y Scheduling:** El problema es decidir cuándo se le puede dar a un proceso determinado acceso a un recurso (un recurso es cualquier objeto, elemento, componente, dato, SC, por la que un proceso puede ser demorado esperando adquirirlo).

**Alocación Shortest-Job-Next:** Varios procesos compiten por el uso de un único recurso compartido. Un proceso requiere el uso del recurso ejecutando request(time,id), donde time es un entero que especifica cuánto va a usar el recurso el proceso, e id es un entero que identifica al proceso que pide. Cuando un proceso ejecuta request, si el recurso está libre, es inmediatamente alocado al proceso; si no, el proceso se demora. Después de usar el recurso, un proceso lo libera ejecutando release( ). Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) que tiene el mínimo valor de time. Si dos o más procesos tienen el mismo valor de time, el recurso es alocado al que ha esperado más.

Por ejemplo, la política SJN puede ser usada para alocación de procesador (en la cual time es el tiempo de ejecución), para spooling de archivos a una impresora (time sería el tiempo de impresión), o para servicio de file trasfer (ftp) remoto (time sería el tiempo estimado de transferencia). La política SJN es atractiva pues minimiza el tiempo promedio de ejecución. Sin embargo, es inherentemente unfair: un proceso puede ser demorado para siempre si hay una corriente continua de requests especificando tiempos de uso menores. (Tal unfairness es extremadamente improbable en la práctica a menos que el recurso esté totalmente sobrecargado. Si interesa unfairness, la política SJN puede ser levemente modificada de modo que un proceso que ha estado demorado un largo tiempo tenga preferencia; esta técnica es llamada “aging”).



Los semáforos b[id] son ejemplos de semáforos privados. A diferencia de los vistos anteriormente, se asocian con un único proceso s es un semáforo privado si exactamente un proceso ejecuta operaciones P sobre s Resultan útiles para señalar procesos individuales.

**CLASE 5 Y 6**

**Monitores:** En semáforos existen variables compartidas globales a los procesos, sentencias de control de acceso a la SC dispersas en el código. Ademas al agregar procesos, se debe verificar acceso correcto a las VC, y aunque EM y SxC son conceptos distintos, se programan de forma similar.

Los ***monitores*** son módulos de programacion con más estructura, y que pueden ser implementados tan eficientemente como los semáforos. Encapsulan las representaciones de objetos (recursos) y brindan 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.

La Exclusion Mutua esta implícita asegurando que los procedures en el mismo monitor no ejecutan concurrentemente. Y la Sincronizacion por condición se realiza de forma explícita con variables condición.

Un proceso que invoca un procedure puede ignorar cómo está implementado. El programador del monitor puede ignorar cómo o dónde se usan los procedures.

Un monitor agrupa la representación y la implementación de un recurso compartido (clase). Lo que distingue a un monitor de un TAD es que puede ser compartido por procesos concurrentes.

Tiene interfaz y cuerpo, La interfaz especifica operaciones (métodos) que brinda el recurso, El cuerpo tiene variables que representan el estado del recurso y procedures que implementan las operaciones de la interfaz.

Sólo los nombres de los procedures son visibles desde afuera. Sintácticamente, los llamados al monitor tienen la forma: call NombreMonitor.operacion (argumentos).

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

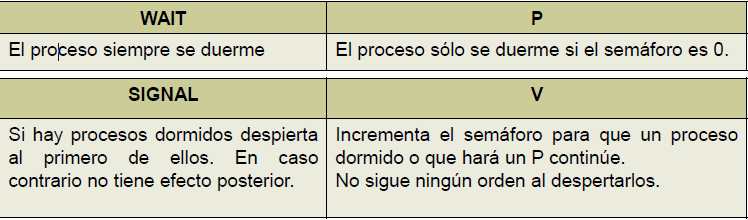
**Soncronizacion:**

Como decía, la sincronización por condición es programada explícitamente con variables condición ***cond c***; el valor asociado a c es una cola de procesos demorados, no visible directamente al programador.

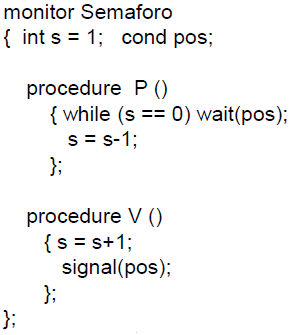
* ***wait(c)*** el proceso se demora al final de la cola de cv y deja el acceso exclusivo al monitor.
* ***signal(c)*** despierta al proceso que está al frente de la cola y lo saca de ella (cola vacía = skip). Ese proceso puede ejecutar cuando readquiera el acceso exclusivo al monitor (depende de la disciplina de señalización)
* ***signal\_all(c)*** despierta todos los procesos demorados en cv. El tiempo en que cada uno reinicie efectivamente la ejecución dependerá de las condiciones de exclusión mutua.

**Signal and continue**: el proceso que ejecuta SIGNAL retiene el control exclusivo del monitor y puede seguir ejecutando, mientras el proceso despertado pasa a competir por acceder nuevamente al monitor y continuar su ejecución en la instrucción siguiente al wait (Unix, Java, Pthreads). ***Este es el que usamos en la practica.***

**Signal and wait**: el proceso que hace SIGNAL pasa a competir por acceder nuevamente al monitor, mientras que el despertado pasa a ejecutar dentro del monitor en la instrucción siguiente al wait.



**Realizar un semáforo con monitores**



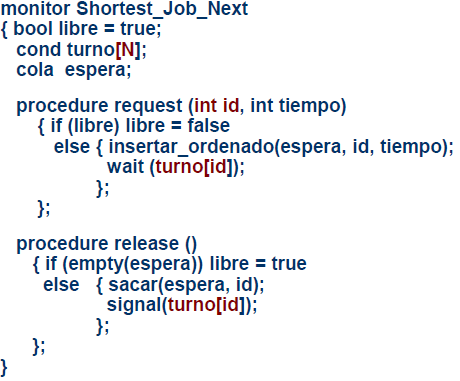
**Passing the condition**: Passing the condition consiste en que un proceso señale una variable de condición si hay procesos demorados en ella, pero sin volver verdadera a la condición asociada; o bien vuelva verdadera la condición sin señalar la variable si no hay procesos demorados. Permite asegurar que al señalar una variable, el proceso despertado pueda continuar ejecutándose. Como el proceso que señala no vuelve verdadera a la condición, si otro proceso tomara control del monitor antes que el despertado, verá falsa la condición y pasará a la cola de espera, siendo el proceso despertado el único que podrá continuar.

**Relacion entre Passing the Baton y Passing the condition:** Passing the baton permite que un proceso pase a otro demorado el control (mutex e) y que éste sea el único que pueda ejecutar secciones exclusivas; passing the condition permite que un proceso avise a otro que están dadas las condiciones para que continúe y que éste sea el único que pueda continuar la ejecución de los que requieran dicha condición.

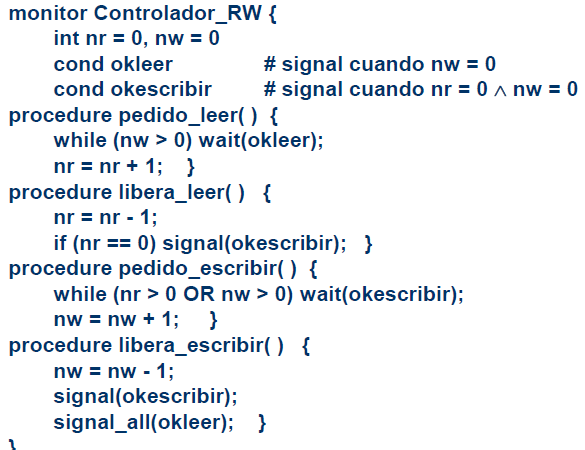
El proceso despertado por passing the baton se ejecutará antes que el resto de los que competían por el mutex; el proceso despertado por passing the condition se ejecutará antes que el resto de los que requieran la condición (tal vez tomen control del monitor, pero se bloquearán).

Aunque passing the baton puede utilizarse para cualquier tipo de sincronización, passing the condition sólo puede aplicarse cuando los procesos que ejecutan wait y signal realizan acciones complementarias entre sí (por ejemplo, uno decrementa un contador y el otro lo incrementa, o uno setea una variable a verdadero y el otro a falso).

**SJN con monitores**



**Lectores y escritores:** El monitor arbitra el acceso a la BD. Los procesos dicen cuándo quieren acceder y cuándo terminaron : requieren un monitor con 4 procedures: pedido\_leer, libera\_leer, pedido\_escribir y libera\_escribir. nr número de lectores y nw número de escritores (variables permanentes).

****

**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.

Solucion: Los clientes y el peluquero son procesos, y la peluquería es un monitor dentro del cual los procesos interactúan. Los clientes son clientes que piden un servicio del peluquero, en este caso cortes de pelo. El peluquero es un servidor que repetidamente provee el servicio. Este tipo de interacción es un ejemplo de relación cliente/servidor.

Para implementar estas interacciones, podemos modelizar la peluquería con un monitor con tres procedures: get\_haircut, get\_next\_customer, y finished\_cut. Los clientes llaman a get\_haircut; luego vuelven de este procedure luego de recibir un corte de pelo. El peluquero repetidamente llama a get\_next\_customer para esperar que un cliente se siente en su silla, luego le corta el pelo y finalmente llama a finished\_cut para permitir que el cliente deje la peluquería. Se usan variables permanentes para registrar el estado de los procesos y para representar las sillas donde los procesos duermen.

Dentro del monitor, necesitamos sincronizar las acciones del peluquero y los clientes. Primero, un peluquero y un cliente necesitan rendezvous; es decir, el peluquero tiene que esperar que llegue un cliente, y un cliente tiene que esperar que el peluquero esté disponible. Segundo, el cliente necesita esperar hasta que el peluquero terminó de cortarle el pelo, lo cual es indicado cuando éste abre la puerta de salida. Finalmente, antes de cerrar la puerta de salida, el peluquero necesita esperar hasta que el cliente haya dejado el negocio. En resumen, tanto el peluquero como el cliente atraviesan una serie de etapas sincronizadas, comenzando con un rendezvous.

Un rendezvous es similar a una barrera de dos procesos pues ambas partes deben arribar antes de que cualquiera pueda seguir. (Sin embargo, difiere de una barrera de dos procesos pues el peluquero puede hacer rendezvous con cualquier cliente). Como con las barreras, la forma más directa de especificar las etapas de sincronización tales como estas es emplear contadores incrementales que registran el número de procesos que alcanzaron cada etapa.

**CLASE 7**

**Arq. de memoria distribuida** 🡪procesadores + memo local + red de comunicaciones + mecanismo de comunicación / sincronización 🡪 intercambio de mensajes

**Programa distribuido** 🡪 programa concurrente comunicado por mensajes. Supone la ejecución sobre una arq. de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida).

**Paradigmas para la interacción entre procesos**

* ***Servidores Replicados***: Varios procesos servidores manejan recursos compartidos, sirviendo solicitudes de múltiples clientes. Puede haber un servidor por cliente o un pool de servidores. Se utilizan si hay múltiples instancias del recurso (de modo que cada servidor administre una) o si hay varios recursos y se desea dar a los clientes la impresión de que sólo hay uno.
* ***Algoritmos Heartbeat***: Procesos que intercambian información haciendo send de sus datos y luego un receive desde sus vecinos. Útil para paralelizar soluciones iterativas (cada worker resuelve un paso, pero precisa los resultados de sus vecinos para el paso siguiente, así que comunica los resultados y espera los de ellos).
* ***Algoritmos Pipeline***: Una serie de procesos conectados en pipe/tubería, por los que fluyen los datos mediante send/receive. Cada uno es un productor/consumidor -filtro-. Puede variar la forma de interconexión (lineal, circular o cerrada -con coordinador de realimentación Wn a W1-), y pueden manejarse distintos datos o funciones en cada proceso.
* **Probes & Echos -sondas y ecos-:** Procesos que trabajan en una topología de grafo. Un proceso puede diseminar/juntar información haciendo sends en paralelo a sus vecinos y esperando que estos lo propaguen, y eventualmente le envíen una respuesta. El recorrido de la sonda es análogo a un DFS del grafo, en paralelo por las distintas ramas. Si no hay información de la topología completa los sends se hacen a todos los vecinos, pero puede mejorarse si se cuenta con esa información calculando antes el árbol mínimo de expansión. En el primer caso se puede aprovechar el eco para construir una representación de la topología. Además, es posible debido a la presencia de ciclos, que un nodo reciba el probe desde más de un vecino; en dicho caso podría sólo responder al primero con la respuesta precisa, y enviar ecos nulos -sin datos de respuesta- al resto.
* **Algoritmos Broadcast**: Permiten diseminar información en una arquitectura distribuida permitiendo tomar decisiones descentralizadas. Puede utilizar una primitiva broadcast disponible en algunas arquitecturas de red -token ring, ethernet-, que encola los mensajes en todos los canales de un arreglo. Como esta operación NO es atómica, distintos procesos podrían recibir mensajes de broadcast con distinto origen en distinto orden.

Sea P[1:n] un arreglo de procesos, y ch[1:n] un arreglo de canales, uno por proceso. Entonces un proceso P[i] hace broadcast de un mensaje m ejecutando broadcast ch(m) 🡪 La ejecución de broadcast pone una copia de m en cada canal ch[i], incluyendo el de P[i]. El efecto es el mismo que ejecutar n sentencias send en paralelo, donde cada una envía m a un canal diferente.

* **Token Passing**: Un token es una clase especial de mensaje que puede ser usado o para dar permiso para tomar una acción o para reunir información de estado global. El envío de información global y/o la toma de determinadas decisiones se basan en el uso de tokens enviados entre los procesos. La topología no necesariamente debe ser de anillo ni los tokens globales.
* **Manager/workers**: Implementación distribuida del modelo de bag of tasks que consiste en un proceso controlador de datos y/o procesos y múltiples workers que acceden a él para poder obtener datos y/o tareas a ejecutar en forma distribuida.

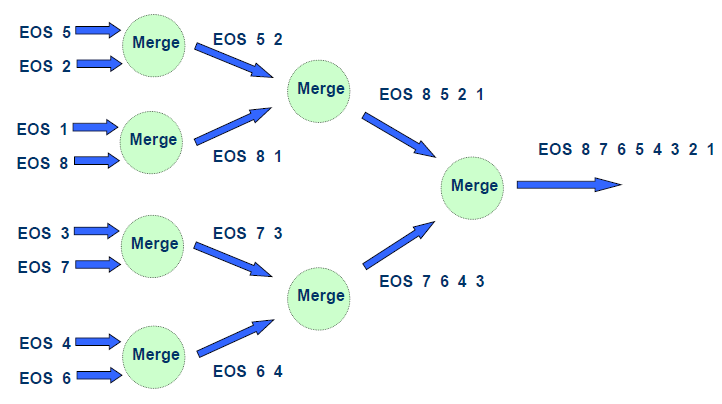
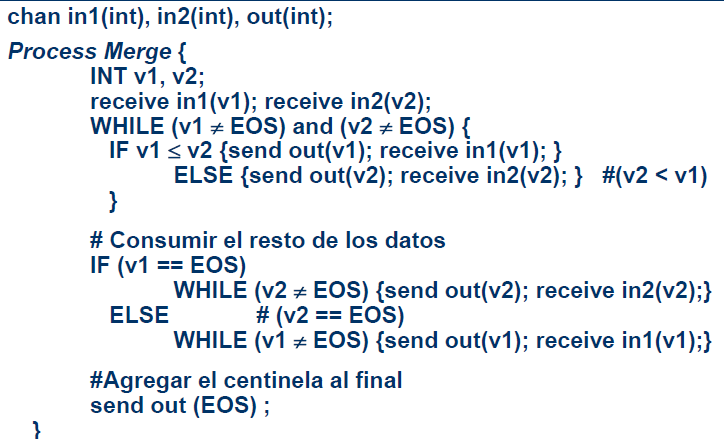
En **problemas de broadcast** sin conocimientos sobre la topología, como solución general sería conveniente Probe & Echo, ya que permite el envío de mensajes a todos los nodos sin importar la arquitectura de red -podría obviarse el echo si no se requieren respuestas, dependiendo del problema. *Los algoritmos de broadcast* podrían utilizarse pero sólo si la primitiva broadcast es soportada para la red en cuestión, siendo preferible en esos casos. *Manager/workers y servidores replicados* no aplican, están orientados a problemas diferentes. *Token passing* serviría para topologías de anillo pero no sería la mejor opción para grafos generales. **Manager/Workers, Heartbeat y Pipeline son más comunes en programas paralelos**. Con pipeline los datos fluyen en una dirección y podría no ser posible hacer broadcast. Con heartbeat podrían compartirse datos con los vecinos y luego de varias iteraciones sería de conocimiento global, pero sin datos de la topología no se sabría cuántas iteraciones son necesarias, y obviamente la convergencia sería más lenta. Además el tipo de problemas en los que se aplican estos algoritmos raramente precisarán broadcasts.

**Pasaje de Mensajes Asincronicos**

* Un proceso agrega un mensaje al final de la cola (“ilimitada”) de un canal ejecutando un send, que no bloquea al emisor.
* Un proceso recibe un mensaje desde un canal con receive, que demora al receptor hasta que en el canal haya al menos un mensaje; luego toma el primero y lo almacena en variables locales.
* Acceso a los contenidos de c/ canal: atómico y respeta 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. Es util cuando el proceso puede hacer trabajo productivo mientras espera un mensaje, pero 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).

**FILTROS:** Proceso que recibe mensajes de uno o más canales de E/ y envía mensajes a uno o más canales de S/. La salida de un filtro es función de su estado inicial y de los valores recibidos. Esta función del filtro puede especificarse por un predicado que relacione los valores de los mensajes de S/ con los de E/.

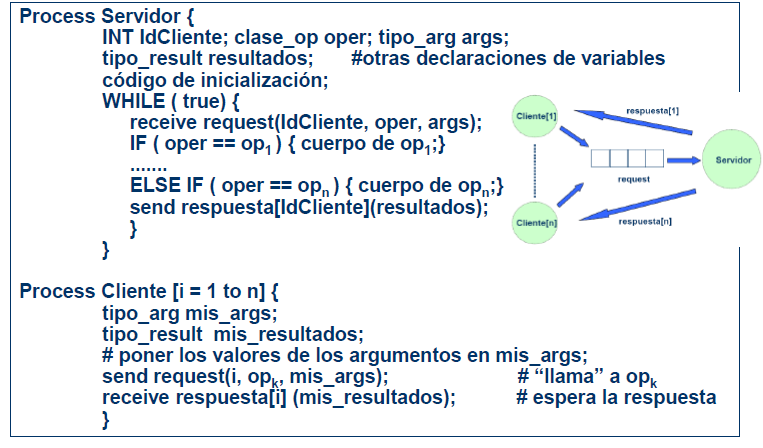
Problema: ordenar una lista de N números de modo ascendente. Podemos pensar en un filtro Sort con un canal de E/ (N números desordenados) y un canal de S/ (N números ordenados). Una solución más eficiente que la “secuencial” es construir una red de pequeños procesos que ejecutan en paralelo e interactúan para “armar” la salida ordenada.



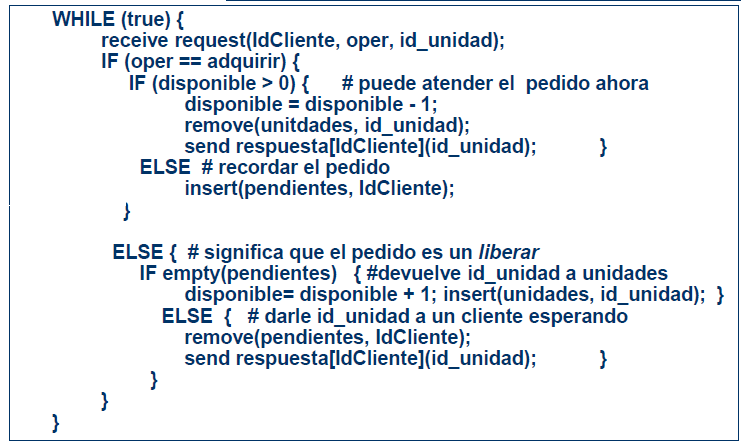
Para realizar esta red de ordenacion, los canales de entrada y salida necesitan ser compartidos. En particular, el canal de salida usado por una instancia de Merge necesita ser el mismo que uno de los canales de entrada usada por la siguiente instancia de Merge en el grafo. Esto puede ser programado de dos maneras. La primera es usar **static naming**: declarar todos los canales como un arreglo global, y que cada instancia de Merge reciba de dos elementos del arreglo y envíe a otro elemento. Esto requiere embeber el árbol en el arreglo para que los canales accedidos por Merge sean una función de i. La segunda forma es usar **dynamic naming**: declarar todos los canales globales, parametrizar los procesos y darle a cada proceso tres canales cuando es creado. Esto hace más fácil la programación de los procesos Merge pues cada uno es textualmente idéntico. Sin embargo, requiere tener un proceso principal que dinámicamente cree y pase canales a los procesos Merge.

**CLIENTES Y SERVIDORES:** Analizamos cómo convertir monitores en servidores y cómo implementar manejadores de recursos compartidos. Existe una dualidad entre monitores y PM: c/u de ellos puede simular al otro.

**Monitor con varias operaciones realizado con PMA**

****

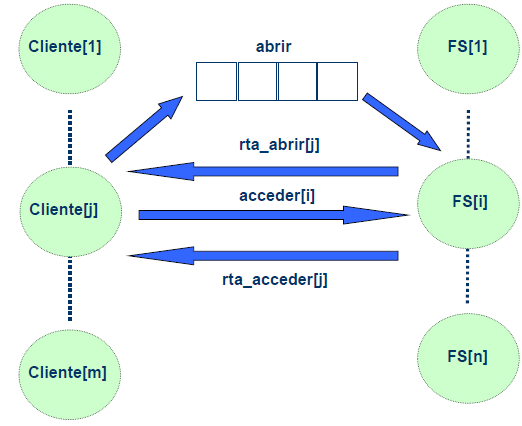
**Monitor con “wait” realizado con PMA**

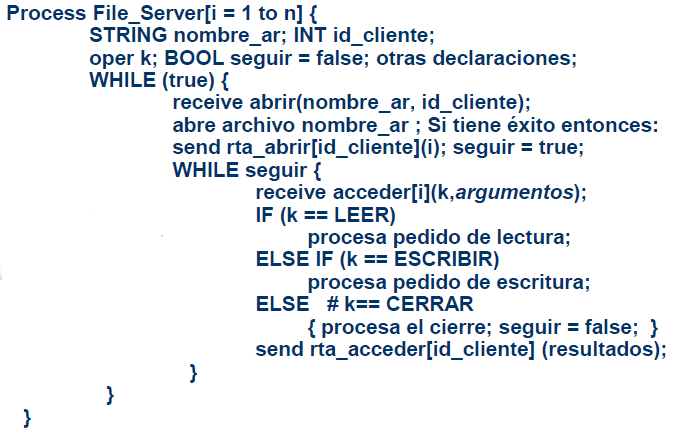
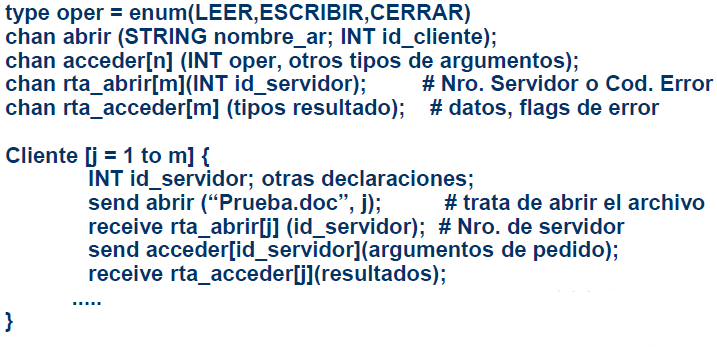
****

La performance relativa de los programas basados en monitor versus en mensajes depende solo de la eficiencia relativa de la implementación de los distintos mecanismos y de la arquitectura física de soporte. En máquinas de memoria compartida, los llamados a procedure y las acciones sobre variables condición tienden a ser más eficientes que las primitivas de MP. Por esta razón, la mayoría de los SO para tales máquinas se basan en una implementación estilo monitor. Por otro lado, la mayoría de los sistemas distribuidos se basan en MP pues es más eficiente y la abstracción apropiada.

**File Servers: Continuidad Conversacional**

Procesos “cliente” que acceden a archivos externos almacenados en disco. Deben hacer OPEN; si el archivo se puede abrir hacen una serie de pedidos de READ o WRITE y luego cierran el archivo (CLOSE). Si hay N archivos, consideramos 1 File Server por archivo. Los procesos servidores son idénticos, y cualquiera de ellos que esté libre puede atender un requerimiento de OPEN.

****

****

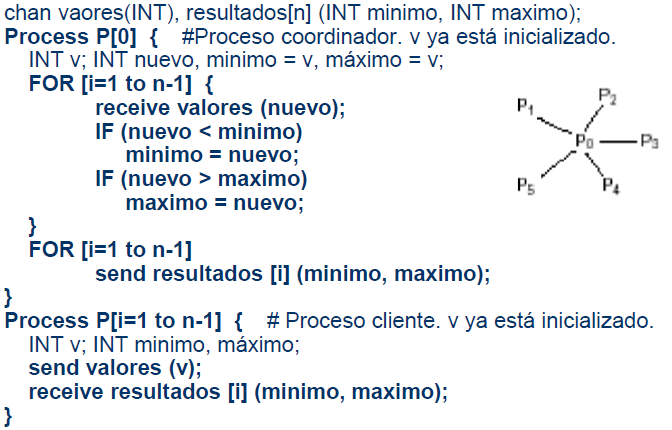
La interacción entre un cliente y un server es un ejemplo de continuidad conversacional. En particular, un cliente comienza una “conversación” con un FS cuando ese server recibe el pedido de apertura. Luego el cliente sigue conversando con el mismo server. Esto se programa haciendo que el server primero reciba de open, luego repetidamente recibe de su elemento de access.

**Pares (peers) interactuantes: Intercambio de valores**

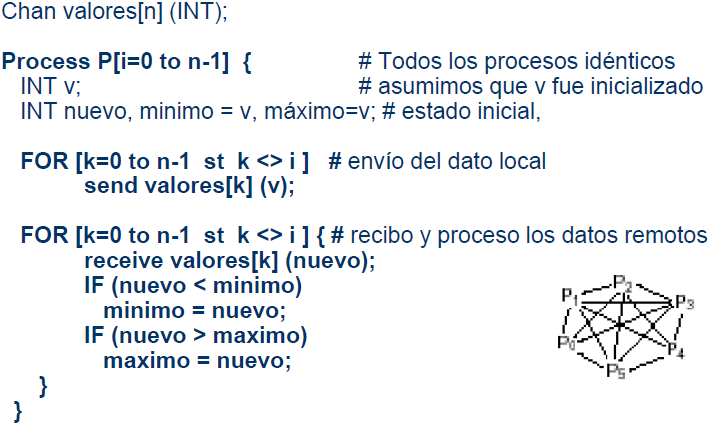
Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: centralizado, simétrico y en anillo circular. Problema: c/ proceso tiene un dato local V y los N procesos deben saber cuál es el menor y cuál el mayor de los valores



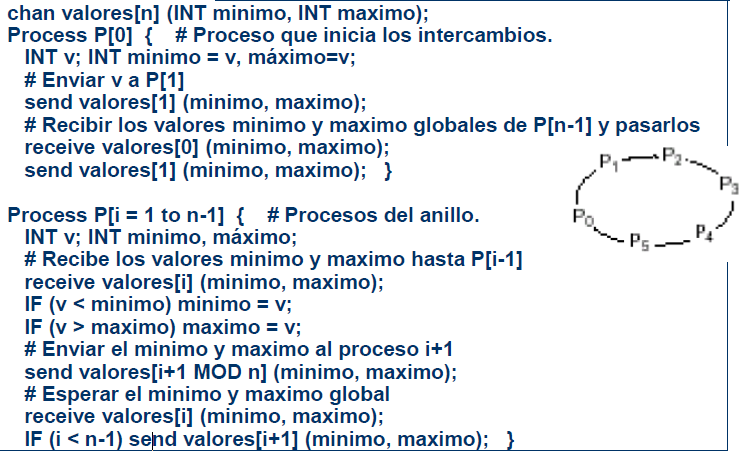
La **arq. centralizada** es apta para una solución en que todos envían su dato local V al procesador central, éste ordena los N datos y reenvía la información del mayor y menor a todos los procesos 🡪 2(N-1) mensajes. Si p[0] dispone de una primitiva broadcast se reduce a N mensajes.



En la **arquitectura simétrica o “full conected”** hay un canal entre cada par de procesos. Todos los procesos ejecutan el mismo algoritmo. Cada proceso trasmite su dato local v a los n-1 restantes procesos. Luego recibe y procesa los n-1 datos que le faltan, de modo que en paralelo toda la arquitectura está calculando el mínimo y el máximo y toda la arquitectura tiene acceso a los n datos. Ejemplo de solución SPMD: cada proceso ejecuta el mismo programa pero trabaja sobre datos distintos n(n-1) mensajes 🡪 Si disponemos de una primitiva de broadcast, serán nuevamente n mensajes.



Un tercer modo de organizar la solución es tener un **anillo** donde P[i] recibe mensajes de P[i-1] y envía mensajes a P[i+1]. P[n-1] tiene como sucesor a P[0]. Esquema de 2 etapas. En la 1ra c/ proceso recibe dos valores y los compara con su valor local, trasmitiendo un máximo local y un mínimo local a su sucesor. En la 2da etapa todos deben recibir la circulación del máximo y el mínimo global. P[0] deberá ser algo diferente para “arrancar” el procesamiento. Se requerirán 2 (n-1) mensajes. Notar que si bien el número de mensajes es lineal (igual que en la centralizada) los tiempos pueden ser muy diferentes.



*Conclusion*: ***Simétrica*** es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast). Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup. ***Centralizada y anillo*** usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance. ***En centralizada***, los msgs al coordinador se envían casi al mismo tiempo 🡪 sólo el 1er receive del coordinador demora mucho. ***En anillo***, todos los procesos son productores y consumidores. El último tiene que esperar a que todos los otros (uno por vez) reciban un msg, hacer poco cómputo, y enviar su resultado. Los mensajes circulan 2 veces completas por el anillo 🡪 Solución inherentemente lineal y lenta para este problema, pero puede funcionar si cada proceso tiene mucho cómputo.

**Algoritmo Heartbeat para el cálculo de la topología de una red**

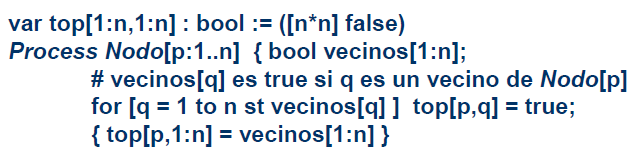
Procesadores conectados por canales bidireccionales. C/ uno se comunica sólo con sus vecinos y conoce esos links Cómo puede cada procesador determinar la topología completa de la red?

Soluciones posibles:

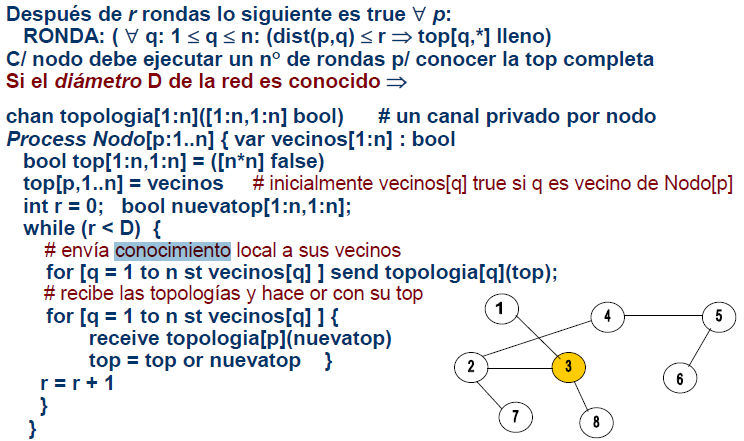
* los procesos tienen acceso a MC
* distribuida: los vecinos interactúan para intercambiar info local

El Algoritmo Heartbeat se expande, enviando información; luego se contrae, incorporando nueva información.

**Variables compartidas (solución poco realista para este tipo de problemas)**:



**Variables globales**

****

**Extensión de lenguajes secuenciales con bibliotecas específicas:** Una técnica muy utilizada es el desarrollo de bibliotecas de funciones que permiten comunicar/sincronizar procesos, no dependientes de un lenguaje de programación determinado. Las soluciones basadas en bibliotecas pueden ser menos eficientes que los lenguajes “reales” de pgmción concurrente, aunque permiten “agregarse” al código secuencial con bajo costo de desarrollo. Las arquitecturas distribuidas han potenciado las soluciones basadas en PVM o MPI, que son básicamente bibliotecas de comunicaciones. Un esquema anterior (y original) es el de LINDA. Los pgms MPI usan un estilo SPMD. C/ proceso ejecuta una copia del mismo programa, y puede tomar distintas acciones de acuerdo a su “identidad”. Las instancias interactúan llamando a funciones MPI, que soportan comunicación proceso-a-proceso y grupales.

**CLASE 8**

**Pasaje de mensajes sincronicos:** La principal diferencia con PMA es que la primitiva de transmisión (llamemosla 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. Los canales son punto a punto (1 emisor – 1 receptor). ***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***.

**CSP** (Communicating Sequential Processes, Hoare 1978) fue uno de los desarrollos fundamentales en Programación Concurrente. Muchos lenguajes reales (OCCAM, ADA, MPD) se basan en CSP. Las ideas básicas introducidas por Hoare fueron PMS y comunicación guardada: PM con waiting selectivo. Un canal es un link directo entre dos procesos en lugar de mailbox global. Son half-duplex y nominados. Las sentencias de Entrada (? o query) y Salida (! o shriek o bang) son el único medio por el cual los procesos se comunican. Para que se produzca la comunicación, deben matchear, y luego se ejecutan simultáneamente. Efecto: sentencia de asignación distribuida.

***Formas generales de las sentencias de comunicación***:

Destino ! port(e1, ..., en);

Fuente ? port(x1, ..., xn);

Destino y Fuente nombran un proceso simple, o un elemento de un arreglo de procesos. Fuente puede nombrar cualquier elemento de un arreglo (Fuente[\*]). port es un canal de comunicación simple en el proceso destino o un elemento de un arreglo de ports en el proceso destino. Los ports se usan p/ distinguir entre distintas clases de mensajes que un proceso podría recibir (puede omitirse si es sólo uno). Dos procesos se comunican cuando ejecutan sentencias de comunicación que hacen matching.

***CSP – Comunicación guardada***

Limitaciones de ? y ! ya que son bloqueantes.

Problema si un proceso quiere comunicarse con otros (quizás por distintos ports) sin conocer el orden en que los otros quieren hacerlo con él.

Las sentencias de comunicación guardada soportan comunicación no determinística:

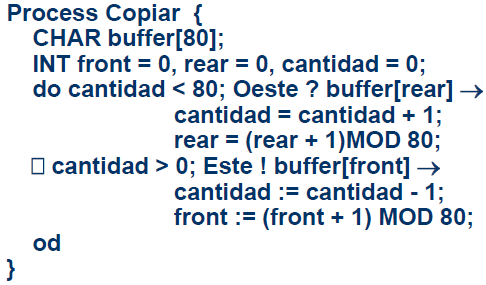
**B; C 🡪 S;**

Aquí B es una expresión booleana opcional, C es una sentencia de comunicación opcional, y S es una lista de sentencias. Si B se omite, tiene el valor implícito de true. Si C se omite, una sentencia de comunicación guardada es simplemente una sentencia guardada.

Juntos, B y C forman la guarda. La guarda tiene éxito (succeds) si B es true y ejecutar C no causaría una demora; es decir, algún otro proceso está esperando en una sentencia de comunicación matching. La guarda falla si B es falsa. La guarda se bloquea si B es true pero C no puede ser ejecutada sin causar demora.

Las sentencias de comunicación guardadas aparecen dentro de sentencias if y do. Ahora una sentencia if es ejecutada como sigue. Si al menos una guarda tiene éxito, una de ellas es elegida no determinísticamente. Primero se ejecuta la sentencia de pasaje de mensajes, y luego la correspondiente lista de sentencias. Si todas las guardas fallan, el if termina. Si ninguna guarda tiene éxito y algunas guardas están bloqueadas, la ejecución se demora hasta que alguna guarda tenga éxito. Dado que las variables no son compartidas, el valor de una expresión booleana en una guarda no puede cambiar hasta que el proceso ejecute sentencias de asignación. Así, una guarda bloqueada no puede tener éxito hasta que algún otro proceso alcanza una sentencia de comunicación matching.

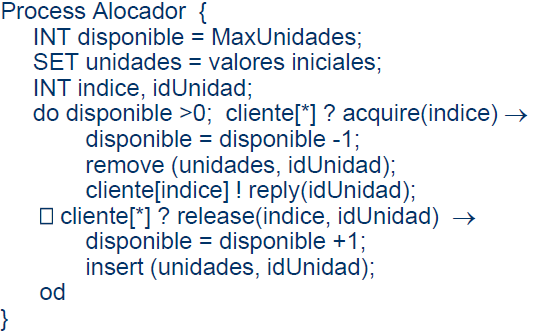
Ejemplo Buffer limitado



Con PMA, procesos como Oeste e Este ejecutan a su propia velocidad pues hay buffering implícito.

Con PMS, es necesario programar un proceso adicional para implementar buffering si es necesario.

**CSP asignación de recursos**



La solución es concisa. Usa múltiples ports y un brazo del do para atender c/u. Se demora en un mensaje aquire hasta que haya unidades, y no es necesario salvar los pedidos pendientes.

**Lenguaje Occam**

Hoare introdujo CSP como lenguaje formal que sigue el modelo de PMS, pero nunca fue implementado en forma completa. OCCAM es un lenguaje real, que implementa lo esencial de CSP sobre la arquitectura “modelo” de los transputers.

Transputers *(multiprocesador de memoria distribuida de bajo costo)* + OCCAM = “sistema multiprocesador p/ procesamiento concurrente”, donde tanto la arquitectura como el lenguaje son simples y adecuadas a la programación concurrente PMS. OCCAM es un lenguaje simple con una sintaxis rígida. Los procesos y los caminos de comunicación entre ellos son estáticos (cantidades fijas definidas en compilación). Modelo de comunicación sincrónico por canales half dúplex.

Las sentencias básicas (asignación y comunicación) son vistas como procesos primitivos.

No soporta recursión, ni creación o nombrado dinámico 🡪 algunos algoritmos son difíciles de programar, aunque el compilador puede determinar cuántos procesos tiene un programa y cómo se comunican. Permite mapear procesos a procesadores del transputer.

LasuUnidades básicas de programa son las declaraciones y 3 "procesos" primitivos: asignación, input (receive), output (sync\_send). Los procesos primitivos se combinan en procesos convencionales usando “constructores”: secuencial (SEQ), paralelo (PAR, similar a la sentencia co), y sentencia de comunicación guardada.

**Programación Paralela con 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.

**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.

**CLASE 9**

**Conceptos de RPC y Rendezvous**

El PM se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la comunicación unidireccional. Para resolver Cliente/Servidor la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas).Además, cada cliente necesita un canal de reply distinto lo que lleva a un gran numero de canales.

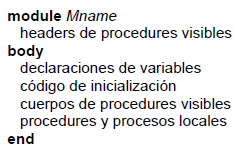
***RPC (Remote Procedure Call) y Rendezvous*** son técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional que son ideales para programar aplicaciones Cliente servidor.

RPC y Rendezvous combinan una interfaz “tipo monitor” con operaciones 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).

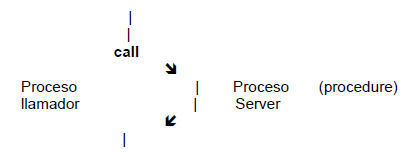
La diferencia entre RPC y rendezvous es la manera en la cual se sirven las invocaciones de operaciones. Una aproximación es declarar un procedure para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado. Esto se llama remote procedure call pues el llamador y el cuerpo del procedure pueden estar en máquinas distintas. La segunda aproximación es rendezvous con un proceso existente. Un rendezvous es servido por medio de una sentencia de entrada (o accept) que espera una invocación, la procesa, y luego retorna resultados. (Suele llamarse rendezvous extendido para contrastarlo con el rendezvous simple entre sentencias de comunicación matching en SMP).

**Remote Procedure Call:** Con RPC, usaremos una componente de programa (el módulo) que contiene tanto procesos como procedures. También, permitiremos que los módulos residan en espacios de direcciones distintos, por ejemplo, en diferentes nodos en una red. Los procesos dentro de un módulo pueden compartir variables y llamar a procedures declarados en ese módulo. Sin embargo, un proceso en un módulo puede comunicarse con procesos en un segundo módulo solo llamando procedures del segundo módulo.

Para distinguir entre procedures que son locales a un módulo y los que proveen canales de comunicación, un módulo tiene dos partes. La parte de especificación (spec) contiene headers de procedures que pueden ser llamados desde otros módulos. El cuerpo implementa estos procedures y opcionalmente contiene variables locales, código de inicialización, y procedures locales y procesos. La forma de un módulo es:



La implementación de un llamado intermódulo es distinta que para un llamado local, pues dos módulos pueden estar en espacios de direcciones diferentes. En particular, un nuevo proceso sirve el llamado. El proceso llamador se demora mientras este proceso server ejecuta el cuerpo del procedure que implementa opname. Cuando el server vuelve de opname, envía los argumentos resultado y cualquier valor de retorno al proceso llamador, luego el server termina. Después de recibir resultados, el proceso llamador continúa. Si el proceso llamador y el procedure están en el mismo espacio de direcciones, con frecuencia es posible evitar crear un nuevo proceso para servir un llamado remoto (el proceso llamador puede temporariamente convertirse en server y ejecutar el cuerpo del procedure. Pero en general, un llamado será remoto, de modo que un proceso server debe ser creado o alocado de un pool preexistente de servers disponibles.



Por sí mismo, RPC es solo un mecanismo de comunicación. Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador (como si éste estuviera ejecutando el llamado 🡪 la sincronización entre ambos es implícita).

Necesitamos que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo). Esto comprende EM y SxC.

Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan con exclusión mutua (un solo proceso por vez) o concurrentemente.

* Si ejecutan con EM las VC son protegidas automáticamente contra acceso concurrente, pero es necesario programar SxC.
* Si pueden ejecutar concurrentemente necesitamos mecanismos para programar EM y SxC (c/ módulo es un programa concurrente)

Podemos usar cualquier método ya descripto (semáforos, monitores, o incluso rendezvous). Es más general asumir que los procesos pueden ejecutar concurrentemente (más eficiente en un multiprocesador de MC).

**Rendezvous:** Rendezvous combina comunicación y sincronización. Como con RPC, un proceso cliente invoca una operación por medio de un call, pero esta operación es servida por un proceso existente en lugar de por uno nuevo. Un proceso servidor usa una sentencia de entrada para esperar por un call y actuar. Las operaciones se atienden una por vez más que concurrentemente

La especificación de un módulo contiene declaraciones de los headers de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones. Si un módulo exporta opname, el proceso server en el módulo realiza rendezvous con un llamador de opname ejecutando una sentencia de entrada:

in opname(identif. formales) 🡪 S; ni

Las partes entre in y ni se llaman operación guardada. Una sentencia de E/ demora al proceso server hasta que haya al menos un llamado pendiente de opname; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta S y finalmente retorna los parámetros de resultado al llamador. Luego, ambos procesos pueden continuar.

**Rendezvous. El lenguaje ADA**

Desde el punto de vista de la concurrencia, un pgm Ada tiene “tasks” (tareas o procesos) que pueden ejecutar independientemente y que contienen primitivas de sincronización. Los puntos de invocación (entrada) a una task se denominan entrys y están especificados en la parte visible (header de la tarea). Una tarea puede decidir si acepta la comunicación con otro proceso, mediante la primitiva accept. Se puede declarar un tipo task, y luego crear instancias de procesos (tareas) identificado con dicho tipo (arreglo, puntero, instancia simple).

El rendezvous es el principal mecanismo de sincronización en Ada y también es el mecanismo de comunicación primario.

Las declaraciones de entry son similares a las de op):

**entry identificador (formales)**

Los parámetros del entry pueden ser IN, OUT o IN OUT.

También soporta arreglos de entries, llamados familias de entry.

Entry call: Si el task T declara el entry E, otras tasks en el alcance de la especificación de T pueden invocar a E con:

**call T.E (parámetros reales)**

La ejecución del call demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).

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

**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.

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

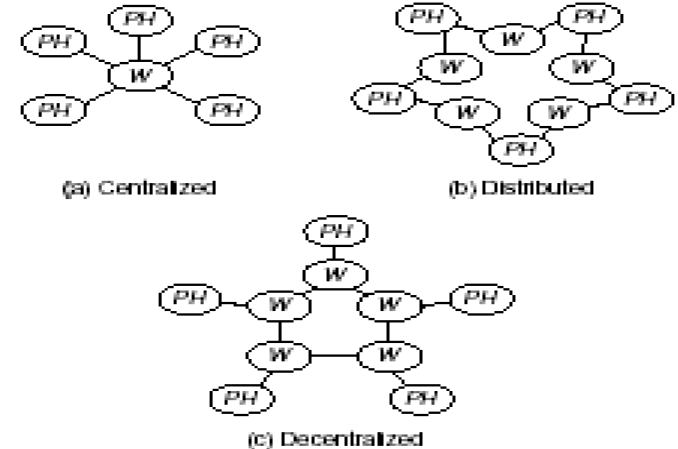
Una operación es servida por un procedure (proc) o por rendezvous (sentencias in). La elección la toma el programador del módulo.

**CLASE 10**

**Filósofos Descentralizado (Servidores replicados):** Un server puede ser replicado cuando hay múltiples instancias de un recurso: c/ server maneja una.

También puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.

1. Un posible modelo de solución al problema de los filósofos es el centralizado: los procesos Filósofo se comunican con UN proceso Mozo que decide el acceso o no a los recursos.
2. Una 2da solución (distribuida) supone 5 procesos Mozo c/u manejando un tenedor. Un Filósofo puede comunicarse con 2 Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos NO se comunican entre ellos.
3. En la 3ra solución (descentralizada), c/ Filósofo ve un único Mozo. Los Mozos se comunican entre ellos (cada uno con sus 2 vecinos) para decidir el manejo del recurso asociado a “su” Filósofo.

****

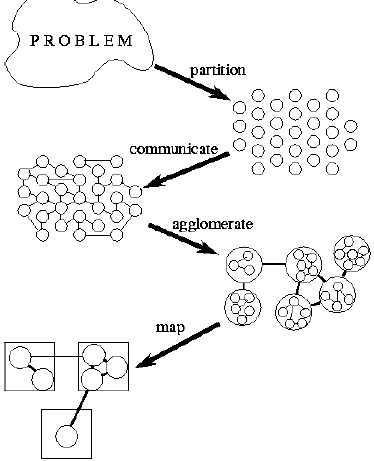
La **solución descentralizada** tiene un mozo por filósofo. El algoritmo usado por los mozos es token passing, donde tokens = tenedores. La solución puede adaptarse para coordinar acceso a archivos replicados, o dar solución eficiente al problema de EM distribuida.Cada tenedor es un token que tiene uno de los dos mozos o está en tránsito entre ellos.Cuando el filósofo quiere comer, le pide a su mozo que adquiera dos tenedores. Si el mozo no los tiene, interactúa con sus mozos vecinos para obtenerlos. Luego mantiene el control mientras el filósofo come. Debe evitarse el deadlock, que podría darse si un mozo necesita dos tenedores y no los puede lograr. La idea básica para evitar deadlock es que un mozo mantenga un tenedor si es necesario y aún no fue usado; en otro caso el mozo lo cede. Específicamente, cuando un filósofo comienza a comer, su mozo marca ambos tenedores como "sucios". Cuando otro mozo quiere un tenedor, si está sucio y no está siendo usado, el primer mozo lo limpia y lo cede. El primer mozo no puede tomar nuevamente el tenedor hasta que no haya sido usado pues solo son pasados tenedores sucios entre los mozos. Sin embargo, un tenedor sucio puede ser reusado hasta que lo necesite otro mozo.

**Programa Paralelo:** Programacion concurrente escrito para resolver un problema en menos tiempo que el secuencial. El objetivo ppal es reducir el tiempo de ejecución, o resolver problemas + grandes o con > precisión en el mismo tiempo.

**Diseño de algoritmos paralelos:** 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)



**Metricas del paralelismo:** En un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance. Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución. A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores (sistema paralelo = algoritmo + arquitectura sobre la que se implementa).

En la medición de performance es usual elegir un problema y testear el tiempo variando el n° de procesadores. Aquí subyacen las nociones de speedup y eficiencia, y la ley de Amdahl. Los modelos de tiempo fijo son una alternativa. Otro tema de interés es la escalabilidad, que da una medida de usar eficientemente un número creciente de procesadores.

* **Tamaño del problema (W):** El tiempo de ejecución paralelo, para un sistema paralelo dado, es función del tamaño del problema y el n° de procesadores (Tp(W,p)).
* **Speedup (S):** S es el cociente entre el tiempo de ejecución serial del algoritmo serial conocido más rápido (Ts) y el tiempo de ejecución paralelo del algoritmo elegido (Tp) S = Ts / Tp. Las diferentes definiciones de tiempo de ejecución serial llevan a distintas definiciones de speedup. Rango de valores: en general entre 0 y p.
* **Eficiencia (E):** Cociente entre speedup y número de procesadores. E = Ts / pTp. Mide la fracción de tiempo en que los procesadores son útiles para el cómputo. El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto. No debieran usarse speedup y eficiencia como métricas independientes del tiempo de corrida.
* **Ley de Amdahl:** Permite calcular el límite superior para el speedup de un algoritmo en particular. En general, un algoritmo cualquiera realiza una entrada, procesamiento y salida. No todo el algoritmo puede mejorarse en una versión paralela, ya que normalmente la etapa de entrada -o inicializaciones- y salida mantendrán componentes secuenciales: sólo el procesamiento será apto de optimización. La mejora total alcanzable en un programa estará limitada entonces por su componente no paralelizable.

  
El SpeedUp máximo alcanzable se obtiene considerando la valor óptimo de un speedup lineal para la parte paralelizable (SpeedUp Parte Paralelizable = P, la cantidad de procesadores).

Como muchas veces la E/S llevan un tiempo mucho menor al procesamiento la limitación puede no ser tan fuerte.

La ley de Amdahl indica que la mejora obtenida al alterar una porción del sistema estará limitada por la fracción del tiempo que dicha porción se utiliza. Así, es el algoritmo y no la cantidad de procesadores el factor que limita el speedup; agregar procesadores puede aumentar el speedup pero sólo hasta cierto punto.

**Granularidad:** Cuando el nro de procesadores crece, normalmente la cantidad de procesamiento en c/u disminuye y las comunicaciones aumentan. Esta relación se conoce como granularidad. Puede definirse la granularidad de una aplicación o una máquina paralela como la relación e/ la cantidad mínima o promedio de operaciones aritmético-lógicas con respecto a la cantidad mínima o promedio de datos que se comunican. La relación cómputo/comunicación impacta en la complejidad de los procesadores: a medida que son más independientes y realizan más operaciones A-L e/ comunicaciones, también deben ser más complejos. Si la granularidad del algoritmo es diferente a la de la arquitectura, normalmente se tendrá pérdida de rendimiento.