Concurrente Resumen 2021

Concurrencia

Escriba definición de Concurrencia, Paralelismo, Procesamiento distribuido

La **concurrencia** es la capacidad de ejecutar múltiples actividades en paralelo o en forma simultánea.
El **paralelismo** es la ejecución concurrente sobre diferentes componentes físicos (procesadores). El paralelismo es un concepto asociado con la existencia de múltiples procesadores ejecutando un algoritmo en forma coordinada y cooperante. Al mismo tiempo se requiere que el algoritmo admita una descomposición en múltiples procesos ejecutables en diferentes procesadores (concurrencia).
El **procesamiento distribuido** es un conjunto de elementos (heterogéneos) de procesamiento que se interconectan por una red de comunicaciones (pasaje de mensajes) y cooperan entre ellos para realizar sus tareas asignadas.

Diferencia programa concurrente, programa paralelo, programa distribuido, programa secuencial

Un **programa concurrente** consiste en un conjunto de tareas o procesos secuenciales que pueden ejecutarse intercalándose en el tiempo y que cooperan para resolver un problema. Básicamente, es un concepto de software que dependiendo de la arquitectura subyacente da lugar a las definiciones de programación paralela o distribuida.
La **programación paralela** consiste en un programa concurrente que se ejecuta sobre múltiples procesadores que pueden tener una memoria compartida y que son utilizados para incrementar la performance de un programa concurrente.
La **programación distribuida** es un caso especial de la programación concurrente en la que se cuenta con varios procesadores pero no se posee una memoria compartida y la comunicación está dada por el pasaje de mensajes.
Un **programa secuencial** mantiene sólo un thread de control, es decir un flujo de control único, administrado por un solo procesador.

Desventajas de la programación concurrente

- **Menor confiabilidad:** En Programación Concurrente los procesos no son completamente independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas.
- **Dificultad para la interpretación y debug:** 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 los procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas.
- **Mayor complejidad en los compiladores y sistemas operativos asociados:** La comunicación y sincronización produce un overhead de tiempo, inútil para el procesamiento. Esto en muchos casos desvirtúa la mejora de performance buscada.
  La mayor complejidad en la especificación de los procesos concurrentes significa que los lenguajes de programación tienen requerimientos adicionales.
- **Mayor costo de los ambientes y herramientas de Ingeniería de Software de sistemas concurrentes:** Aumenta el tiempo de desarrollo y puesta a punto respecto de los programas secuenciales. También puede aumentar el costo de los errores.
- La **paralelización** de algoritmos secuenciales NO es un proceso directo, que resulte fácil de automatizar.
  Para obtener una real mejora de performance, se requiere **adaptar el software concurrente al hardware paralelo**.

Tres grandes clases de aplicaciones concurrentes

El primer tipo de aplicaciones se corresponde cuando ejecutamos N procesos independientes sobre M procesadores, con N > M. Un sistema de software de "multithreading" maneja simultáneamente tareas independientes, asignando (por ejemplo por tiempos) los procesadores de que dispone.
Ejemplos típicos:

- Sistemas de ventanas en PCs o WS.
- Time sharing en sistemas operativos multiprocesador.
- Sistemas de tiempo real en plantas industriales

El 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.
Ejemplos típicos:

- Servidores de archivos (recursos) en una red.
- Sistemas de Bases de datos distribuidas (bancos, reservas de vuelos).
- Servidores WEB distribuidos.
- Arquitecturas cliente-servidor.

El procesamiento paralelo: Se trata de resolver un problema en el menor tiempo posible, utilizando una arquitectura multiprocesador en la que se pueda distribuir la tarea global en tareas que puedan ejecutarse en diferentes procesadores.
Paralelismo de datos y paralelismo de procesos.
Ejemplos típicos:

- Cálculo científico. Modelos de sistemas (meteorología, movimiento planetario).
- Gráficos, procesamiento de imágenes, realidad virtual, procesamiento de video.
- Problemas combinatorios y de optimización lineal y no lineal. Modelos econométricos

Paradigmas de resolución de programas concurrentes

En el **Paralelismo iterativo** un programa tiene un conjunto de procesos (posiblemente idénticos) cada uno de los cuáles tiene uno o más loops. Es decir cada proceso es un programa iterativo.
La idea es que si estos procesos cooperan para resolver un único problema (ejemplo un sistema de ecuaciones) pueden trabajar independientemente y sincronizar por memoria compartida o envío de mensajes. Ejemplo clásico es la multiplicación de matrices.

En el **Paralelismo recursivo** el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos.
Ejemplos clásicos son por ejemplo el sorting by merging o el cálculo de raíces en funciones continuas.

El esquema **productor-consumidor** muestra procesos que se comunican. Es habitual que estos procesos se organicen en pipes a través de los cuáles 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. Distintos niveles de SO.

Cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido. Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Naturalmente unos y otros pueden 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. Ejemplo FS.

En los **esquemas de pares que interactúan**, los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea. El esquema permite mayor grado de asincronismo que C-S. Ejemplo multiplicación de matrices distribuida.

Defina el concepto de no determinismo. Ejemplifique

Los programas concurrentes son **No Determinísticos**: no se puede determinar que para los mismos datos de entrada se ejecute la misma secuencia de instrucciones, tampoco se puede determinar si dará la misma salida.
Sólo se puede asegurar un **Orden Parcial**, esto quiere decir que si tenemos dos procesos concurrentes “p” y “q”, y estos fueron divididos en tres instrucciones cada uno, podemos asegurar que la instrucción pi se ejecutará antes que la instrucción pj si es que i es menor a j, y lo mismo para el proceso q.
Ejemplo x=0 //P y x=x+1 //Q. Escenario 1: P Q x=1, Escenario 2: Q P x=0.

¿Qué significa el problema de “interferencia” en programación concurrente? ¿Cómo puede evitarse?

La interferencia se da cuando un proceso toma una acción que invalida alguna suposición hecha por otro proceso y esto se debe a que las acciones de los procesos en un programa concurrente pueden ser intercaladas. La interferencia se da por la realización de asignaciones en un proceso a variables compartidas que pueden afectar el comportamiento o invalidar un supuesto realizado por otro proceso.
Se puede evitar utilizando mecanismos de sincronización (exclusión mutua o por condición) y también asegurando propiedades como la de ASV (a lo sumo una vez).

Sincronización

Definición Sincronización. Formas y Mecanismos de sincronización

Sincronización: puede definirse como el conocimiento de información acerca de otro proceso para coordinar actividades.

Debemos considerar que:

- El estado de un programa concurrente en cualquier punto de tiempo, consta de los valores de las variables de programa.
- Un programa concurrente comienza la ejecución en algún estado inicial.
- Cada proceso en el programa se ejecuta a una velocidad desconocida.
- A medida que ejecuta, un proceso transforma el estado ejecutando sentencias.
    - Cada sentencia consiste en una secuencia de una o más acciones atómicas que hacen transformaciones de estado indivisibles.
- La ejecución de un programa concurrente genera una secuencia de acciones atómicas que es algún interleaving (entrelazado) de las secuencias de acciones atómicas de cada proceso componente.
- El trace de una ejecución particular de un programa concurrente puede verse como una historia.

El rol de la sincronización es restringir las posibles historias de un programa concurrente a aquellas que son deseables.

Formas (Mecanismos) de sincronización

Exclusión mutua: consiste en asegurar que las secciones críticas de sentencias que acceden a recursos compartidos no se ejecutan al mismo tiempo. La exclusión mutua concierne a combinar las acciones atómicas fine-grained que son implementadas directamente por hardware en secciones críticas que deben ser atómicas para que su ejecución no sea intercalada con otras secciones críticas que referencian las mismas variables.
Sincronización por condición: asegura que un proceso se demora si es necesario hasta que sea verdadera una condición dada. La sincronización por condición concierne a la demora de un proceso hasta que el estado conduzca a una ejecución posterior.

Mecanismos de comunicación y sincronización entre procesos

Memoria compartida: Los procesos intercambian mensajes sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella. Lógicamente los procesos no pueden operar simultáneamente sobre la memoria compartida, lo que obligará a BLOQUEAR y LIBERAR el acceso a la memoria. La solución más elemental será una variable de control tipo “semáforo” que habilite o no el acceso de un proceso a la memoria compartida.

En **memoria distribuida** los procesos también pueden comunicarse a través de mensajes que llevan datos. Para esto es necesario establecer un canal lógico o físico para transmitir información entre procesos. El pasaje de mensajes puede ser sincrónico o asincrónico y es independiente de la arquitectura.
Para el pasaje de mensajes el lenguaje debe proveer un protocolo adecuado. Para que la comunicación sea efectiva los procesos deben “saber” cuando tienen mensajes para leer y cuando deben transmitir mensajes.

Compare en términos de facilidad de programación

La resolución de problemas con memoria compartida requiere el uso de exclusion mutua o sincronización por condición para evitar interferencias entre los procesos mientras que con memoria distribuida la información no es compartida si no que cada procesador tiene su propia memoria local y para poder compartir información se requiere del intercambio de mensajes evitando así problemas de inconsistencia.
Por lo tanto, resulta mucho más fácil programar con memoria distribuida porque el programador puede olvidarse de la exclusión mutua y muchas veces de la necesidad de sincronización básica entre los procesos que es provista por los mecanismos de pasajes de mensajes. Igualmente la facilidad de programación con uno o con otro modelo también está dada por el problema particular que deba ser resuelto.

¿En un programa concurrente pueden estar presentes más de un mecanismo de sincronización?

Si. Por ejemplo, un programa que tiene dos procesos, uno productor y otro consumidor, que comparten un sector de memoria para el pasaje de los datos.
Se usaría exclusión mutua para que no accedan a la memoria compartida al mismo tiempo, y sincronización por condición para que un mensaje no sea recibido antes que enviado, y para que un mensaje no sobreescriba a uno que todavía no fue recibido.

Cuáles son los defectos que presenta la sincronización por busy waiting? Diferencie esta situación respecto de los semáforos.

Problemas busy waiting: Los protocolos de sincronización que usan solo busy waiting pueden ser difíciles de diseñar, entender y probar su corrección. La mayoría de estos protocolos son bastante complejos, no hay clara separación entre las variables usadas para sincronización y las usadas para computar resultados.
Otra deficiencia es que es ineficiente cuando los procesos son implementados por multiprogramación. Un procesador que está ejecutando un proceso “spinning” podría ser empleado más productivamente por otro proceso. Esto también ocurre en un multiprocesador pues usualmente hay más procesos que procesadores.

Busy waiting y semáforos son herramientas que sirven para sincronización pero son distintas ya que semáforos evitan los problemas que tiene busy waiting. Al bloquearse en un semáforo un proceso no consume tiempo de procesamiento hasta que no tenga posibilidad de ejecutarse, en cuyo caso, es puesto en la cola de listos para poder usar el procesador.

¿Qué significa que un problema sea de “exclusión mutua selectiva”?

Que un problema sea de exclusión mutua selectiva significa que los procesos tienen que competir por sus recursos no con todos los demás procesos sino con un subconjunto de ellos.

El problema de los lectores y escritores, ¿es de exclusión mutua selectiva? ¿Por qué?

El problema de los lectores y escritores es de exclusión mutua selectiva porque existen distintas clases de procesos que compiten por el acceso a la BD que es compartida por ambos. Los procesos escritores individualmente compiten por el acceso con cada uno de los otros y los procesos lectores como una clase compiten con los escritores.

Si en el problema de los lectores-escritores se acepta sólo 1 escritor o 1 lector en la BD, ¿tenemos un problema de exclusión mutua selectiva? ¿Por qué?

- Si en el problema se acepta solo 1 escritor y 1 lector en la BD el problema deja de ser de exclusión mutua selectiva porque la competencia por acceder a la BD es contra todos.

Si en lugar de 5 filósofos fueran 3, ¿el problema seguiría siendo de exclusión mutua selectiva? ¿Por qué?

- En el caso de que fueran 3 filósofos y no 5 como en la definición del problema general, no sería de exclusión mutua selectiva ya que un proceso compite con todos los demás procesos y no solo con un subconjunto de ellos, dado que el resto de los procesos son sus adyacentes.

El problema de los filósofos resuelto de forma centralizada y sin posiciones fijas ¿es de exclusión mutua selectiva? ¿Por qué?

- Sin posiciones fijas se refiere a que cada filósofo solicita cubiertos y no necesariamente los de sus adyacentes, el coordinador o mozo se los da si es que hay dos cubiertos disponibles sin tener en cuenta vecinos, el problema no sería de exclusión mutua selectiva ya que competirían entre todos por poder acceder a los tenedores para poder comer

Propiedades, Fairness, Atomicidad

Defina el concepto de granularidad. ¿Qué relación existe entre la granularidad de programas y de procesos?

Se puede definir como granularidad a la relación que existe entre el procesamiento y la comunicación.

- En una **arquitectura de grano fino** existen muchos procesadores pero con poca capacidad de procesamiento por lo que son mejores para programas que requieran mucha comunicación. Entonces, los programas de grano fino son los apropiados para ejecutarse sobre esta arquitectura ya que se dividen en muchas tareas o procesos que requieren de poco procesamiento y mucha comunicación.
- Por otro lado, en una **arquitectura de grano grueso** se tienen pocos procesadores con mucha capacidad de procesamiento por lo que son más adecuados para programas de grano grueso en los cuales se tienen pocos procesos que realizan mucho procesamiento y requieren de menos comunicación.

¿Qué se entiende por arquitectura de grano grueso? ¿Es más adecuada para programas con mucha o poca comunicación?

Cuando hablamos de arquitectura de grano grueso decimos, que se tratan de pocos procesadores muy poderosos (que realizan mucho cálculo). Dicha arquitectura es más adecuada para programas con poca comunicación.

Atomicidad de Grano Grueso y Atomicidad de Grano Fino

Acción atómica de grano grueso: hace una transformación de estado indivisible. Esto significa que cualquier estado intermedio que podría existir en la implementación de la acción no debe ser visible para los otros procesos.
Acción atómica de grano fino: es implementada directamente por el hardware sobre el que ejecuta el programa concurrente.

¿En qué consiste el problema de la sección crítica?

El problema de la SC consiste en desarrollar un protocolo de E/S a la SC para poder ejecutar una porción de código en forma atómica.

Propiedad de programa. Propiedades de vida y seguridad. Ejemplifique.

Propiedad de programa es un atributo que es verdadero en cualquier posible historia del programa, y por lo tanto de todas las ejecuciones del programa. Cada propiedad puede ser formulada en términos de dos clases especiales de propiedades:

- **Propiedad de seguridad** asegura que el programa nunca entra en un estado malo (es decir uno en el que algunas variables tienen valores indeseables).
  Ejemplos de propiedad de seguridad: **Ausencia de demora innecesaria**, **Exclusión mutua** y **Ausencia de deadlock**

- **Propiedad de vida** es aquella que asegura que algo bueno eventualmente ocurre durante la ejecución. Son ejemplos de propiedad de vida que un proceso eventualmente alcance su SC o que un mensaje llegue a destino pero estas propiedades dependen de la política de scheduling.
  Ejemplos de propiedad de vida: **Terminación** y **Eventual Entrada**

Ausencia de demora innecesaria (Propiedad de Seguridad): si un proceso está intentando entrar en su sección crítica y no hay otro en la sección crítica, ni queriendo entrar, el primero no debería estar bloqueado.grained

bool in1 = false, in2 = false
MUTEX: ¬(in1 ^ in2)

process SC1{
  while (true) {
  <await (not in2) in1 = true>
  SC
  SNC
  in1 = false;
  }
}
	process SC2{
	  while (true) {
	  <await (not in1) in2 = true>
	  SC
	  SNC
	  in2 = false;
	  }
	}

En este caso, no se cumple porque la liberación (el paso de in1/in2 a false) está después de la SNC.

Exclusión mutua (Propiedad de Seguridad): solo un proceso está ejecutando su sección crítica a la vez

int var = 0

process SC1{
  while (true) {
  var = 1 (SC)
  SNC
  }
}
	process SC2{
	  while (true) {
	  var = 2 (SC)
	  SNC
	  }
	}

En este caso no hay mecanismos de exclusión sobre la SC.

Eventual entrada (Propiedad de Vida): un proceso que quiere entrar a su sección crítica lo hará eventualmente

bool in1 = **true**, in2 = false
MUTEX: ¬(in1 ^ in2)

process SC1{
  while (true) {
  <await (not in2) in1 = true>
  SC
  SNC
  }
}
	process SC2{
	  while (true) {
	  <await (not in1) in2 = true>
	  SC
	  in2 = false;
	  SNC
	  }
	}

Como in1 comienza en true y SC1 no cambia su estado, por lo tanto se bloquea el ingreso de SC2.

Ausencia de deadlocks (Propiedad de Seguridad): Si uno o más procesos están intentando entrar a su SC al menos uno de ellos tendrá éxito

bool in1=**true**, in2=**true**
MUTEX: ¬(in1 ^ in2)

process SC1{
  while (true) {
  <await (not in2) in1 = true>
  SC
  in1 = false;
  SNC
  }
}
	process SC2{
	  while (true) {
	  <await (not in1) in2 = true>
	  SC
	  in2 = false;
	  SNC
	  }
	}

Como in1 e in2 comienzan en true, ninguno de los procesos puede ingresar a la SC.

Algoritmos para la resolución de Sección Crítica

Solución grano fino: "Spin Locks"
Esta solución tiene como objetivo **hacer atómico el await de grano grueso,** para esto se **usa alguna instrucción especial** que casi todas las máquinas tienen, que puede usarse para implementar las acciones atómicas condicionales de este programa (test-and-set, fetch-and-add, compare-and-swap). Por ahora definimos y usamos Test-and-Set (TS)

La instrucción TS toma dos argumentos booleanos: un lock compartido y un código de condición local cc. Como una acción atómica, TS setea cc al valor de lock, luego setea lock a true:

TS(lock,cc): ⟨ cc := lock; lock := true ⟩
Cuando se usa una variable de lockeo de este modo, se lo llama spin lock pues los procesos “dan vueltas” (spin) mientras esperan que se libere el lock.

Esta solución 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).

Aunque la última solución es correcta, se demostró que en multiprocesadores puede llevar a **baja performance si varios procesos están compitiendo por el acceso a una SC**. Esto es porque lock es una variable compartida y todo proceso demorado continuamente la referencia. Esto causa “memory contention”, lo que degrada la performance de las unidades de memoria y las redes de interconexión procesador-memoria. Además, la instrucción TS escribe en lock cada vez que es ejecutada, aún cuando el valor de lock no cambie.
Spin locks no controla el orden de los procesos demorados entonces es posible que alguno no entre nunca si el scheduling no es fuertemente fair (race conditions).

Algoritmo TIE-BREAKER
El algoritmo tie-breaker (o algoritmo de Peterson) es un protocolo de SC (Sección Crítica) que requiere solo **scheduling incondicionalmente fair** para satisfacer la propiedad de eventual entrada. Además **no requiere** instrucciones especiales del tipo **Test-and-Set.** Sin embargo, el algoritmo es mucho más complejo que la solución spin lock. **Usa una variable adicional para romper empates, indicando qué proceso fue el último en comenzar a ejecutar su protocolo de entrada a la SC**. Si hay n procesos, el entry protocol en cada proceso consiste de un loop que itera a través de n-1 etapas. En cada etapa, usamos instancias del algoritmo tie-breaker para dos procesos para determinar cuáles procesos avanzan a la siguiente etapa. El algoritmo tie-breaker n-proceso es **costoso en tiempo y bastante complejo y difícil de entender**. Esto es en parte porque no es obvio cómo generalizar el algoritmo de 2 procesos a n.

Algoritmo Ticket
Es una solución para N-procesos **más fácil de entender** que la solución de N procesos del algoritmo Tie-Breaker. La solución también ilustra cómo pueden usarse contadores enteros para ordenar procesos. **En esta solución 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. La ausencia de deadlock y de demora innecesaria resultan de que los valores de turno son únicos. Con **scheduling débilmente fair se asegura eventual entrada.** El algoritmo ticket tiene un problema potencial que es común en algoritmos que emplean incrementos en contadores: los valores de number y next son ilimitados. **Si el algoritmo corre un tiempo largo, se puede alcanzar un overflow. Para este algoritmo podemos resolver el problema reseteando los contadores a un valor chico (digamos 1) cada vez que sean demasiado grandes**. Si el valor más grande es al menos tan grande como n, entonces los valores de turn[i] se garantiza que son únicos.

```
var number := 1, next := 1, turn[1:n] : int := ( [n] 0 )
P[i: 1..n] ::
        do true →
                turn[i] := FA(number,1)
                do turn[i] ≠ next → skip od
                critical section
                next := next + 1
                non-critical section
        od
```

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

FA(var,incr): ⟨ temp := var; var := var + incr; return(temp) ⟩
La mayor fuente de demora en el algoritmo ticket es esperar a que turn[i] sea igual a next.

Algoritmo Bakery
El algoritmo ticket puede ser implementado directamente en máquinas que tienen una instrucción como Fetch-and-Add. Si solo tenemos disponibles instrucciones menos poderosas, podemos simular la parte de obtención del número del algoritmo ticket usando algún algoritmo SC con busy waiting (como Spin lock). Pero eso requiere usar otro protocolo de SC, y la solución podría no ser fair. Bakery algorithm e**s un algoritmo del tipo de ticket que es fair y no requiere instrucciones de máquina especiales**. El **algoritmo es más complejo** que el ticket, pero ilustra una manera de romper empates cuando dos procesos obtienen el mismo número**. No requiere**

un contador global próximo 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.

El algoritmo asegura entrada eventual si el scheduling es débilmente fair pues una vez que una condición de demora se convierte en true, permanece true.

Propiedad de "A lo sumo una vez" (<= 1)

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

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

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

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

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

int x=0, y=0;	No hay ref. críticas en ningún proceso.
co x=x+1 // y=y+1 oc;	En todas las historias x = 1 e y = 1
int x = 0, y = 0;	El 1er proceso tiene 1 ref. crítica. El 2do ninguna.
co x=y+1 // y=y+1 oc;	Siempre y = 1 y x = 1 o 2
int x = 0, y = 0;	Ninguna asignación satisface ASV.
co x=y+1 // y=x+1 oc;	Posibles resultados: x= 1 e y =2 / x = 2 e y = 1

Nunca debería ocurrir x = 1 e y = 1 → ERROR

Si hay una referencia crítica no se lo puede referenciar.

Fairness. Relacione dicho concepto con las políticas de scheduling

La mayoría de las propiedades de vida dependen de fairness, la cual trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los otros procesos. Cuando hay varios procesos hay varias acciones atómicas elegibles (es elegible si es la próxima acción atómica en el proceso que será ejecutado). Una política de scheduling determina cuál será la próxima en ejecutarse.

- **Fairness incondicional:** Una política de scheduling es incondicionalmente fair si toda acción **atómica incondicional** que es elegible eventualmente es ejecutada.
- **Fairness débil:** Una política de scheduling es débilmente fair si es incondicionalmente fair y toda **acción atómica condicional** que se vuelve elegible eventualmente es ejecutada si su guarda se convierte en true y de allí **en adelante permanece true**.
- **Fairness fuerte:** Una política de scheduling es fuertemente fair si es incondicionalmente fair y toda **acción atómica condicional** que se vuelve elegible eventualmente es ejecutada si su guarda es **true con infinita frecuencia**.

Barreras

Defina el concepto de sincronización barrier. ¿Cuál es su utilidad?

Una barrera es un punto de demora a la que deben llegar todos los procesos antes de permitirles pasar y continuar su ejecución.
Dependiendo de la aplicación las barreras pueden necesitar reutilizarse más de una vez (por ejemplo en algoritmos iterativos).
Como muchos problemas pueden ser resueltos con algoritmos iterativos paralelos en los que cada iteración depende de los resultados de la iteración previa, es necesario proveer sincronización entre los procesos al final de cada iteración, para realizar esto se utilizan las barreras.

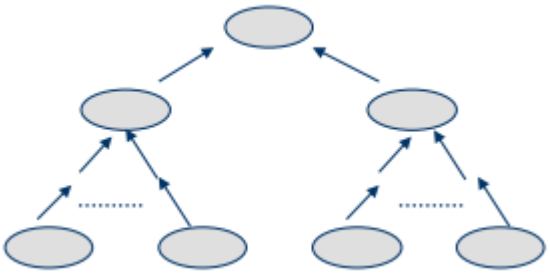
¿Qué es una barrera simétrica?

Una barrera simétrica es un conjunto de barreras entre pares de procesos que utilizan sincronización barrier. En cada etapa los pares de procesos que interactúan van cambiando dependiendo de algún criterio establecido.

Describa combining tree barrier y butterfly barrier. Marque ventajas y desventajas de cada una.

En un combining tree barrier los procesos se organizan en forma de árbol y cumplen diferentes roles. Básicamente, los procesos envían el aviso de llegada a la barrera hacia arriba en el árbol y la señal de continuar cuando todos arribaron es enviada de arriba hacia abajo. Esto hace que cada proceso debe combinar los resultados de sus hijos y luego se los pase a su padre.
Con butterfly barrier lo que se hace es que en cada etapa diferente (son log2 n etapas) cada proceso se sincroniza con uno distinto. Es decir, si s es la etapa cada proceso sincroniza con uno a distancia 2s-1. Así al final de las log2 n etapas cada proceso habrá sincronizado directa o indirectamente con el resto de los procesos.

Combining tree barrier
Ventajas: Su implementación es más sencilla.
Desventajas: Los procesos no son simétricos ya que cada uno cumple diferentes roles, por lo que los nodos centrales realizarán más trabajo que los nodos hoja y la raíz.



Butterfly barrier
Ventajas: La implementación de sus procesos es simétrica ya que todos realizan la misma tarea en cada etapa que es la de sincronizar con un par a distancia 2s-1.
Desventajas: Su implementación es más compleja que la del combining tree barrier y además cuando n no es par puede usarse un n próximo par para sustituir a los procesos perdidos en cada etapa; sin embargo esta implementación no es eficiente por lo que se usa una variante llamada Dissemination barrier.

En qué consiste la sincronización barrier? Mencione alguna de las soluciones posibles usando variables compartidas

Consiste en un punto de demora al final de cada iteración que hace de barrera y a la cual deben llegar todos los procesos antes de permitirles continuar.
Ejemplo:

```
Int cantidad=0;
Process Worker[i=1 to n]{
        While(true){
                #Código para implementar la tarea i;
                 <cantidad=cantidad+1> #Puede implementarse con un FA
                 <await(cantidad==n);> #Puede implementarse como while(cantidad!=n) skip;
        }
}
```

Otra solución consiste en distribuir la variable cantidad usando n variables (arreglo arribo[1..n]). El await pasaría a ser <await(arribo[1]+…+arribo[n] == n);>

Explique la semántica de la instrucción de grano grueso AWAIT y su relación con instrucciones tipo Test & Set o Fetch & Add.

Es una acción atómica de grano grueso, la cual es una secuencia de acciones atómicas de grano fino que aparecen como indivisibles. La sentencia **await** es muy poderosa ya que puede ser usada para especificar acciones atómicas arbitrarias de grano grueso. Esto la hace conveniente para expresar sincronización.
Este poder expresivo también hace a await muy costosa de implementar en su forma más general. Sin embargo, hay casos en que puede ser implementada eficientemente utilizando instrucciones especiales que pueden usarse para implementar las acciones atómicas condicionales (test-and-set, fetch-and-add, compare-and-swap).
La relación entre la instrucción de grano grueso AWAIT y las instrucciones T&S y F&A es que estas últimas están presentes en casi todos los procesadores y son utilizadas para hacer atómico el AWAIT de grano grueso implementando los protocolos de E/S de la sección crítica con los que se asegura la atomicidad del AWAIT.

La instrucción Test-and-Set (TS):

```
TS(lock): < cc := lock; lock := true; return cc; >
```

La instrucción TS toma dos argumentos booleanos: un lock compartido y un código de condición local cc. Como una acción atómica, TS setea cc al valor de lock, luego setea lock a true:

TS(lock,cc): 〈 cc := lock; lock := true 〉
Cuando se usa una variable de lockeo de este modo, se lo llama spin lock pues los procesos “dan vueltas” (spin) mientras esperan que se libere el lock.

La instrucción Fetch-and-Add (FA):

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

Tipos de await

Especificamos sincronización por medio de la sentencia await:

- <await (B) S;>

La expresión booleana B especifica una condición de demora; S es una secuencia de sentencias que se garantiza que termina. Una sentencia await se encierra en **corchetes angulares** para indicar que es ejecutada como una acción atómica. En particular, se garantiza que B es true cuando comienza la ejecución de S, y ningún estado interno de S es visible para los otros procesos.

Para especificar sólo exclusión mutua, abreviamos una sentencia await como sigue:

- <S>

Para especificar solo sincronización por condición, abreviaremos una sentencia await como:

- <await (B) >

busy waiting o spinning: Si una sentencia await cumple los requerimientos de la propiedad de a lo sumo una vez, entonces <await B> puede ser implementado como:

- do (not B) skip od

Cuando la sincronización se implementa de esta manera, un proceso se dice que está en busy waiting o spinning, ya que está ocupado haciendo un chequeo de la guarda

Diferencie acciones atómicas condicionales e incondicionales

Acción atómica incondicional: es una que no contiene una condición de demora B. Tal acción puede ejecutarse inmediatamente.
Acción atómica condicional: es una sentencia await con una guarda B. Tal acción no puede ejecutarse hasta que B sea true. Si B es false, solo puede volverse true como resultado de acciones tomadas por otros procesos.

¿En qué consiste la técnica de "Passing the Baton"? ¿Cuál es su utilidad? Aplique este concepto a la resolución del problema de lectores y escritores. ¿Qué relación encuentra con la técnica de "Passing the Condition"?

La técnica consiste en implementar sincronización por condición arbitraria. Cuando un proceso está dentro de una SC mantiene el baton (testimonio, token) que significa permiso para ejecutar. **Cuando el proceso llega a un SIGNAL, pasa el baton (control) a otro proceso.** Si algún proceso está esperando una condición que ahora es verdadera el baton pasa a tal proceso, el cual ejecuta su SC y pasa el baton a otro proceso. **Si ningún proceso está esperando una condición que sea true, el baton se pasa al próximo proceso que trata de entrar a su SC por primera vez.**
Podemos usar semáforos binarios divididos para implementar tanto la Exclusión Mutua como la sincronización por condición. Sea (e) un semáforo binario cuyo valor inicial es 1 que se usa para controlar la entrada a sentencias atómicas y asociamos un semáforo bj y un contador dj cada uno con guarda semánticamente diferente Bj. El semáforo bj se usa para demorar procesos esperando que Bj se convierta en true y dj es un contador del número de procesos demorados sobre bj.

```
Para representar <Si>: P(e)
        Si;
        SIGNAL
Para representar <await (Bj) Sj>: P(e)
if (not Bj) {dj := dj + 1; V(e); P(bj);}                                    Sj;
                                        SIGNAL

En ambos fragmentos, SIGNAL es la siguiente sentencia:
SIGNAL: if (B1 and d1 > 0){ d1 := d1 - 1; V(b1)}
                [] ...
                [] (Bn and dn > 0){ dn := dn - 1; V(bn)}
                        [] else V(e);
                           fi
```

Las primeras N guardas en SIGNAL chequean si hay algún proceso esperando por una condición que ahora es true. Para el caso que no haya ningún proceso esperando por alguna condición verdadera se ejecutará la guarda else donde el semáforo de entrar (e) es señalizado.
Su utilidad es proveer exclusión y controlar cuál proceso demorado es el próximo en seguir y el orden en el cual los procesos son demorados. Puede usarse para implementar cualquier sentencia await tanto <Si> como <await (Bj) Sj >.

Aplicando esta técnica al problema de los lectores y escritores tenemos esta solución:

```
int numLectoresActivos(nr)      sem accesoBaseDeDatos(e)
int numEscritoresActivos(nw)    sem lectores(r)
int lectoresDemorados(dr)       sem escritores(w)
int escritoresDemorados(dw)

process Lector [i=1 to M] {
    while (true) {
        P(accesoBaseDeDatos);
        if(numEscritoresActivos > 0) {
            lectoresDemorados = lectoresDemorados + 1;
            V(accesoBaseDeDatos);
            P(lectores);
        }
        numLectoresActivos = numLectoresActivos + 1;
        if (lectoresDemorados > 0) {
            lectoresDemorados = lectoresDemorados - 1;
            V(lectores);
        }
        else V(accesoBaseDeDatos);
        Lee BD;
        P(accesoBaseDeDatos);
        numLectoresActivos = numLectoresActivos - 1;
        if (numLectoresActivos == 0 and escritoresDemorados > 0) {
            escritoresDemorados = escritoresDemorados - 1;
            V(escritores);
        }
        else V(accesoBaseDeDatos);
    }
}
```

Proceso lector
#Si hay un escritor en la BD me “encolo” para ser el próximo
#Incremento la variable para avisar que hay un lector esperando
#Incremento la variable para avisar que hay un lector en la BD
#Si hay un lector esperando lo despierto
#Sino libero la SC y entro a la BD
#Decremento la variable para avisar que salí de la BD
#Si no hay ningún lector por ingresar y hay escritores dormidos
#Despierto a uno

```
process Escritor [j=1 to N] {
    while(true) {
        P(accesoBaseDeDatos);
        if (numLectoresActivos > 0 or numEscritoresActivos > 0) {
            escritoresDemorados = escritoresDemorados + 1;
            V(accesoBaseDeDatos);
            P(escritores);
        }
        numEscritoresActivos = numEscritoresActivos + 1;
        V(accesoBaseDeDatos);
        Escribe BD;
        P(accesoBaseDeDatos);
        numEscritoresActivos = numEscritoresActivos - 1;
        if (lectoresDemorados > 0) {
            lectoresDemorados = lectoresDemorados - 1;
            V(lectores);
        }
        elseif (escritoresDemorados > 0) {
            escritoresDemorados = escritoresDemorados - 1;
            V(escritores);
        }
        else V(accesoBaseDeDatos);
    }
}
```

Proceso Escritor
#Si hay lectores o escritores en la BD me duermo
#Incremento la variable para avisar que hay un escritor en la BD
#Decremento la variable para avisar que salí de la SC
#Si hay un lector esperando para entrar lo despierto
#Si hay un escritor esperando para entrar lo despierto

La relación que tiene **Passing the Baton con Passing de Condition** es que ambos determinan cual es el proceso demorado que será despertado para que ejecute su SC.
Como **diferencia** entre las técnicas podemos decir de que la técnica Passing the Condition cuando un proceso hace signal pasa el control directamente al proceso que estaba esperando para poder entrar a la SC antes que otros procesos que llaman a un procedure dentro del monitor tengan chance de avanzar, a diferencia de la técnica de Passing the Baton que cuando un proceso hace signal es "globalmente visible" por todos los procesos que quieren entrar en la SC.
La técnica Passing the Condition pasa una condición directamente a un proceso despertado en lugar de hacerla globalmente visible.

Semáforos

Instancia de un tipo de datos abstracto (o un objeto) con sólo 2 operaciones (métodos) atómicas: P y V.
Internamente el valor de un semáforo es un entero 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.
Si la implementación de la demora por operaciones **P** se produce sobre una **cola**, las operaciones son **fair**

Inconvenientes presentan los semáforos como herramienta de sincronización para la resolución de problemas concurrentes

Semáforos Inconvenientes:

- Variables compartidas globales a los procesos
- Sentencias de control de acceso a la SC dispersas en el código
- Al agregar procesos, se debe verificar el acceso correcto a las variables compartidas.
- Aunque Exclusión Mutua y Sincronización por Condición son conceptos distintos, se programan de forma similar

Defina el problema general de alocación de recursos y su resolución mediante una política SJN (Shortest Job Next). ¿Minimiza el tiempo promedio de espera? ¿Es fair? Si no lo es, plantee una alternativa que lo sea.

El problema de la alocación de recursos consiste en decidir cuando se le puede dar a un proceso acceso a un recurso. Un recurso es cualquier cosa por la que un proceso podría ser demorado esperando adquirirlo. Esto incluye entrada a una sección crítica, acceso a una base de datos, una región de memoria, uso de una impresora.

La resolución mediante SJN teniendo en cuenta que existe solo una unidad del recurso compartido consiste en ejecutar request(time, id) cuando un proceso requiere el uso del recurso, 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; sino el proceso se demora. Después de usar el recurso, un proceso lo libera ejecutando release(). Cuando el recurso es liberado, se aloca para el proceso demorado (si lo hay) que tiene el mínimo valor de time. Si 2 o más procesos tienen el mismo valor de time, el recurso es alocado al que ha esperado más.

- La política SJN minimiza el tiempo promedio de ejecución.
- No es fair, es unfair porque un proceso puede ser demorado para siempre si hay un flujo continuo de request especificando tiempos de uso menores.
- La política SJN puede ser modificada para que sea fair de modo que un proceso que ha estado demorado un largo tiempo tenga preferencia; esta técnica se llama 'aging'.

Monitores

Los monitores son módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.

Mecanismo de abstracción de datos:

- encapsulan las representaciones de objetos (recursos).
- 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.

Explicar la forma en que se realiza la comunicación y la sincronización (por exclusión mutua y por condición) entre los procesos con esta herramienta.

La comunicación y la sincronización por exclusión Mutua se realiza de manera implícita asegurando que los procedures en el mismo monitor no se ejecutan concurrentemente.
Por otro lado, la sincronización por Condición se realiza de manera explícita con el uso de variables condición.

Explicar la diferencia entre los protocolos de sincronización en monitores: signal and wait (S&W) y signal and continue (S&C).

Signal and Continue: el proceso que hace el signal continua usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al wait).
Signal and Wait: el proceso que hace el signal pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al wait.

Implementar un monitor que funcione correctamente con el protocolo S&W y no con S&C (además de implementar el monitor enuncie que debería resolver).

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

```
        Bool libre=true;
        Cond turno;
Procedure request {
                If not libre wait (turno, tiempo);
                Libre = false;}
Procedure release {
                Libre = true;
                Signal (turno) ;}
}
```

Con S&C un proceso que es despertado para poder seguir ejecutando es pasado a la cola de ready en cuyo caso su orden de ejecución depende de la política que se utilice para ordenar los procesos en dicha cola. Puede ser que sea retrasado en esa cola permitiendo que otro proceso ejecute en el monitor antes que el por lo que podría no cumplirse el objetivo del SJN.

En cambio, con S&W se asegura que el proceso despertado es el próximo en ejecutar después de que el señalador ejecuta signal. Por lo tanto, SJN funcionaria correctamente de esta forma evitando que cualquier otro proceso listo para ejecutar le robe el acceso al proceso despertado.

Sea la siguiente solución propuesta al problema de alocacion LJN:

```
Monitor LJN {
        Bool libre = true;
        Cond turno;
        Procedure request (int, tiempo){
                If (not libre) wait (turno, (Maxvalor - tiempo))
                Libre = false;}
        Procedure release {
                Libre = true;
                Signal (turno)}}
```

¿Funciona correctamente con disciplina de señalización Signal and continue? Justifique. ¿Funciona correctamente con disciplina de señalización Signal and Wait? Justifique.

Pasa lo mismo que en SJN. La única manera de hacer que ambos funcionen correctamente con S&C es cambiar la implementación y usar passing the condition.

Pthreads

Thread: 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).

POSIX Threads, commonly known as pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API. POSIX Threads is an API defined by the standard POSIX.1c

Indicar y explicar cómo funcionan las herramientas que tiene la librería Pthreads para manejar la sincronización por exclusión mutua y por condición.

Para manejar la sincronización por exclusión mutua la libreria posee la herramienta de mutex locks (pthread_mutex), esta es una variable que tiene 2 estados posibles (bloqueado y desbloqueado), y solo un thread puede bloquear el mutex, ya que la operación Lock es atómica. Para poder entrar a una sección crítica un thread debe haber bloqueado el mutex.
Para manejar la sincronización por condición utiliza como herramienta las variables condición (pthread_cond), estas están asociadas a un predicado que cuando se convierte en true da señal a todos los threads que están esperando en la misma. Cada variable condición siempre tiene un mutex asociado, el cual es bloqueado por cada thread para testear el predicado definido en la variable (cuando es falso el thread espera en la cond).

Suponga la siguiente porción de código con varios hilos que actúan como productores o consumidores. ¿Para qué la sentencia pthread_cond_wait necesita como segundo parámetro una variable mutex? ¿Cómo se asegura que el buffer se accede con exclusión mutua, suponiendo que hayElemento se inicializa en 0?

```
void *productor(void *datos) {
  tipo_element elem;

  while (true) {
     generar_elemento(elem);
     pthread_mutex_lock (&mutex);
     while (hayElemento = = 1)
        pthread_cond_wait (&vacio, &mutex);
     buffer = elem;
     hayElemento = 1;
     pthread_cond_signal (&lleno);
     pthread_mutex_unlock (&mutex);
  }
}
```

predicado

```
void *consumidor(void *datos) {
  tipo_element elem;

  while (true) {
     pthread_mutex_lock (&mutex);
     while (hayElemento = = 0)
        pthread_cond_wait (&lleno, &mutex);
     elem= Buffer;
     hayElemento = 0;
     pthread_cond_signal (&vacio);
     pthread_mutex_unlock (&mutex);
     procesar_elemento(elem);
  }
}
```

pthread_cond_wait tiene una variable mutex obligatoriamente, que los thread bloquean para poder preguntar por el predicado.
Se asegura la exclusión mutua por el uso de la variable pthread_mutex (&mutex)

https://docs.google.com/document/d/1FyYMzavu7ZhcczCeSFbszj0aQnhPP18lYBNHAbiwx6o/edit#

PMA (Pasaje de mensajes asincrónicos)

Analice conceptualmente los modelos de mensajes sincrónicos y asincrónicos. Compárelos en términos de concurrencia y facilidad de programación
Con **pasaje de mensajes asincrónico (PMA)** los canales de comunicación son colas ilimitadas de mensajes. Un proceso agrega un mensaje al final de la cola de un canal ejecutando una sentencia send. Dado que la cola conceptualmente es ilimitada la ejecución de send no bloquea al emisor.
Un proceso recibe un mensaje desde un canal ejecutando la sentencia receive qué es bloqueante, es decir, el proceso no hace nada hasta recibir un mensaje en el canal. La ejecución del receive demora al receptor hasta que el canal esta no vacío, luego el mensaje al frente del canal es removido y almacenado en variables locales al receptor.
El acceso a los contenidos de cada canal es atómico y se respeta el orden FIFO, es decir, los mensajes serán recibidos en el orden en que fueron enviados. Se supone que los mensajes no se pierden ni se modifican y que todo mensaje enviado en algún momento puede ser leído. **Los procesos comparten los canales.** Es ideal para algoritmos del tipo HeartBeat o Broadcast.
En el **pasaje de mensajes sincrónico (PMS)** tanto send cómo receive son primitivas bloqueantes. Si un proceso trata de enviar a un canal se demora hasta que otro proceso está esperando recibir por ese canal. De esta manera, un emisor y un receptor se sincronizan en todo punto de comunicación.
La cola de mensajes asociada a un send sobre un canal se reduce a un mensaje. Esto significa menor memoria. **Los procesos no comparten los canales.** El grado de concurrencia se reduce respecto a PMA. Como contrapartida los casos de falla y recuperación de errores son más fáciles de manejar. Con PMS se tiene mayor probabilidad de deadlock. El programador debe ser cuidadoso para que todos los send y receive hagan matching. Es ideal para algoritmos del tipo Cliente/Servidor o de Filtros.

Librerías para Pasaje de Mensajes: Explicar los 4 diferentes tipos de SEND en que se clasifican las comunicaciones punto a punto en las librerías de Pasaje de Mensajes en general (no se refiere particularmente a MPI). Indicar cuales se corresponden con las herramientas PMA y PMS.

4 tipos de send son: Bloqueante, No bloqueante, Con buffering, Sin buffering
Send bloqueante: El transmisor queda esperando que el mensaje sea recibido por el receptor (PMS)
Send no bloqueante: El transmisor no queda esperando que el mensaje sea recibido por el receptor (PMA)
Send con buffering: La cola de mensajes asociada con el canal del send puede acumular mensajes (PMA)
Send sin buffering: La cola de mensajes asociada con el canal del send se reduce a 1 (PMS)

PMS (Pasaje de mensajes sincrónicos)

¿En qué consiste la comunicación guardada (introducida por CSP) y cuál es su utilidad?

Un proceso puede tener que realizar una comunicación solo si se da una condición o también se puede dar el caso que se quiere comunicar con uno o más procesos y no sabe el orden en el cual otros procesos podrían querer comunicarse con él. Por esto la comunicación no determinística es soportada en forma elegante extendiendo las sentencias guardadas para incluir sentencias de comunicación.
Una sentencia de comunicación guardada tiene la forma B; C → S donde 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** si B es true y ejecutar C no causaría una demora.
La guarda **falla** si B es falsa.
La guarda se **bloquea** si B es true pero C no puede ser ejecutada sin causar demora.

Por ejemplo podemos programar Copy para implementar un buffer limitado. Por ejemplo, lo siguiente bufferea hasta 10 caracteres:

```
Copy:: var buffer[1:10] : char
        var front := 1, rear := 1, count := 0
        do count < 10; West ? buffer[rear]
                count := count + 1;
                rear := (rear mod 10) + 1
         [] count > 0; East ! buffer[front]
                count := count - 1;
                front := (front mod 10) + 1
        od
```

Nótese que la interface a West (el productor) e East (el consumidor) no cambió. La primera tiene éxito si hay lugar en el buffer y West está listo para sacar un carácter; la segunda tiene éxito si el buffer contiene un carácter e East está listo para tomarlo. La sentencia do no termina nunca pues ambas guardas nunca pueden fallar al mismo tiempo (al menos una de las expresiones booleanas en las guardas es siempre true).

Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas.

Sea la sentencia de alternativa con comunicación guardada de la forma:
If B1; comunicacion1 -> S1
□ B2; comunicacion2 -> S2
Fi
El problema de la SC consiste en desarrollar un protocolo de E/S a la SC para poder ejecutar una porción de código en forma atómica.

Primero, se evalúan las expresiones booleanas, Bi y la sentencia de comunicación.

- Si todas las guardas fallan (una guarda falla si B es false), el if termina sin efecto.
- Si al menos una guarda tiene éxito, se elige una de ellas (no deterministicamente).
- Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito (B es true pero la ejecución de la sentencia de comunicación no puede ejecutarse inmediatamente).

Segundo, luego de elegir una guarda exitosa se ejecuta la sentencia de comunicación asociada.
Tercero, y último, se ejecutan las sentencias S relacionadas.

La ejecución de la iteración es similar realizando los pasos anteriores hasta que todas las guardas fallen.

Paradigmas de Interacción entre Procesos

1. **Manager/workers:** Implementación distribuida del modelo de bag of tasks que consiste en un proceso controlador de datos y/o procesos y múltiples procesadores que acceden a él para poder obtener datos y/o tareas para ejecutarlos en forma distribuida.
2. **Algoritmos de heartbeat:** Los procesos periódicamente deben intercambiar información y para hacerlo ejecutan dos etapas; en la primera se expande enviando información (SEND a todos) y en la segunda se contrae adquieren información (RECEIVE de todos). Su uso más importante es paralelizar soluciones iterativas. Ejemplos de problemas que se pueden resolver son computación de grillas (labeling de imágenes) o autómatas celulares (el juego de la vida).
3. **Algoritmos de pipeline:** La información recorre una serie de procesos utilizando alguna forma de receive/send donde la salida de un proceso es la entrada del siguiente. Ejemplos son las redes de filtros o tratamiento de imágenes.
4. **Algoritmos probe/echo:** La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información. Puede usarse para realizar un broadcast (sin spinning tree) o conocer la topología de una red cuando no se conocen de antemano la cantidad de nodos activos.
5. **Algoritmos de Broadcast:** Permiten alcanzar una información global en una arquitectura distribuida. Sirven para la toma de decisiones descentralizadas y para resolver problemas de sincronización distribuida. Un ejemplo típico es la sincronización de relojes en un Sistema Distribuido de Tiempo Real.
6. **Algoritmos Token passing:** En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. Permite realizar exclusión mutua distribuida y la toma de decisiones distribuidas. Un ejemplo podría ser el de determinar la terminación de un proceso en una arquitectura distribuida cuando no puede decidirse localmente.
7. **Servidores replicados:** Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos. Su uso tiene el propósito de incrementar la accesibilidad de datos o servicios donde cada servidor descentralizado interactúa con los demás para darles la sensación a los clientes de que existe un único servidor. Un ejemplo de servidores DNS.
8. **Divide y vencerás:** hace referencia a ir dividiendo un problema una y otra vez, recursivamente, para ir formando problemas más chicos y más fáciles o rápidos de resolver, una vez resueltos se van juntando estas soluciones para resolver el problema grande. Un ejemplo de utilización es encontrar el mínimo de un arreglo de números.
9. El **paradigma SPMD** separa los datos para ejecutarlos paralelamente en varios procesadores, cada uno utiliza el mismo programa pero diferentes datos, un ejemplo de uso es el cálculo de matrices. (No son procesos del todo independientes, cada tanto deben interactuar).

Explique el concepto de broadcast y sus dificultades de implementación en un ambiente distribuido con mensajes sincrónicos y asincrónicos.

El concepto de broadcast consiste en enviar concurrentemente un mismo mensaje a varios procesos.
En la mayoría de las LAN, los procesadores comparten un canal de comunicación común tal como un Ethernet o token ring. Tales redes soportan una primitiva especial llamada broadcast, la cual transmite un mensaje de un procesador a todos los otros. Podemos utilizar broadcast para diseminar o reunir información.

- Con PMA ejecutar broadcast es equivalente a enviar concurrentemente el mismo mensaje a varios canales. Todas las copias de mensajes se encolan y cada copia puede ser recibida más tarde por uno de los procesos participantes. Necesitamos la respuesta para confirmar la llegada de un mensaje, no se garantiza la llegada de un mensaje ante fallas.
- Con PMS por la naturaleza bloqueante de las sentencias de salida, un broadcast sincrónico debería bloquear al emisor hasta que los procesos receptores hayan recibido el mensaje. Hay dos maneras de obtener el efecto de broadcast con PMS. Una es usar un proceso separado para simular un canal broadcast; es decir, los clientes enviarían mensajes broadcast y los recibirían de ese proceso.

  La segunda manera es usar comunicación guardada. Cuando un proceso quiere tanto enviar como recibir mensajes broadcast puede usar comunicación guardada con dos guardas. Cada guarda tiene un cuantificador para enumerar los otros partners. Una guarda saca el mensaje hacia los otros partners; la otra guarda ingresa un mensaje desde todos los otros partners.

¿En qué consiste la utilización de relojes lógicos para resolver problemas de sincronización distribuida? Ejemplifique.

Las acciones de comunicación, tanto send cómo receive, afectan la ejecución de otros procesos por lo tanto son eventos relevantes dentro de un programa distribuido y por su semántica debe existir un orden entre las acciones de comunicación. Básicamente cuando se envía un mensaje y luego este es recibido, existe un orden entre estos eventos ya que el send se ejecuta antes que el receive por lo que puede imponerse un orden entre los eventos de todos los procesos. Sea un ejemplo el caso en que procesos solicitan acceso a un recurso, muchos de ellos podrían realizar la solicitud casi al mismo tiempo y el servidor podría recibir las solicitudes desordenadas por lo que no sabría quién fue el primero en solicitar el acceso y no tendría forma de responder a los pedidos en orden. Para solucionar este problema se podrían usar los relojes lógicos.

Para establecer un orden entre eventos es que se utilizan los relojes lógicos que se asocian mediante un timestamps a cada evento. Entonces, un reloj lógico es un contador que es incrementado cuando ocurre un evento dentro de un proceso ya que el reloj es local a cada proceso y se actualiza con la información distribuida del tiempo que va obteniendo en la recepción de mensajes. Este reloj lógico (rl) será actualizado de la siguiente forma dependiendo del evento que suceda:
Cuando el proceso realiza un SEND, setea el timestamp del mensaje al valor actual de rl y luego lo incrementa en 1.
Cuando el proceso realiza un RECEIVE con un timestamp (ts), setea rl como max (rl, ts+1) y luego incrementa rl.
Con los relojes lógicos se puede imponer un orden parcial ya que podría haber dos mensajes con el mismo timestamp pero se puede obtener un ordenamiento total si existe una forma de identificar unívocamente a un proceso de forma tal que si ocurre un empate entre los timestamp primero ocurre el que proviene de un proceso con menor identificador.

Bag of Tasks

Se parte del concepto de 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);
}
```

Este enfoque puede usarse para resolver problemas con un n° fijo de tareas y para 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.

¿Cuál (o cuáles) es el paradigma de interacción entre procesos más adecuado para resolver problemas del tipo "Juego de la vida"?

El paradigma que mejor se adecua es el heartbeat ya que permite enviar información a todos los vecinos y luego recopilar la información de todos ellos. Entonces una célula recopila la información de sus vecinos en la cual se basa su próximo cambio de estado.

¿Considera que es conveniente utilizar mensajes sincrónicos o asincrónicos?

Es mejor la utilización de mensajes asincrónicos ya que evitan problemas de deadlock cuando se debe decidir qué proceso envía primero su información. Como en el algoritmo general de heartbeat todos los procesos primero envían su información y luego recopilan la información de sus vecinos. Se hace intuitivo el uso de PMA para su resolución evitando tener que realizar procesos asimétricos con demora innecesaria aunque la performance podría mejorar si se utiliza comunicación guardada para las sentencias de envío y recepción.

¿Cuál es la arquitectura de hardware que se ajusta mejor? Justifique claramente sus respuestas.

Es conveniente una arquitectura de grano fino, con mucha capacidad para la comunicación y poca capacidad para el cómputo ya que este tipo de programas es de grano fino, muchos procesos o tareas con poco cómputo que requieren de mucha comunicación. Una arquitectura en forma de grilla es una forma óptima para este tipo de problemas.

MPI - Message Passing Interface

Describa los mecanismos de comunicación y sincronización provistos por MPI, JAVA.

MPI es una biblioteca de comunicaciones a través de pasaje de mensajes que permite comunicar y sincronizar procesos secuenciales escritos en diferentes lenguajes que se ejecutan sobre una arquitectura distribuida. El estilo de programación es SPMD. Cada 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 comunicaciones proceso a proceso y grupales.

En informática , SPMD (programa único, datos múltiples) es una técnica empleada para lograr el paralelismo; es una subcategoría de MIMD . Las tareas se dividen y se ejecutan simultáneamente en varios procesadores con diferentes entradas para obtener resultados más rápidamente. SPMD es el estilo más común de programación paralela. También es un requisito previo para la investigación de conceptos como mensajes activos y memoria compartida distribuida .

- MPI_init: inicializar entorno MPI.
- MPI_finalize: cerrar entorno MPI.
- MPI_common_size y MPI_common_rank: cantidad de procesos en el comunicador e identificador del proceso dentro del comunicador.
- MPI_send y MPI_recv: ambos bloqueantes.
- MPI_Isend y MPI_Irecv: no bloqueantes.

Java soporta concurrencia mediante threads, utiliza métodos sincronizados (monitores). Exclusión mutua implícita provista por la declaración de la palabra "synchronized". Exclusión mutua explícita (sincronización por condición) con los métodos wait, notify y notifyAll. Además provee el uso de RPC en programas distribuidos mediante la invocación de métodos remotos (RMI).

El server y los clientes pueden residir en máquinas diferentes. Una aplicación que usa RMI tiene 3 componentes:

- Una interfaz que declara los headers para métodos remotos
- Una clase server que implementa la interfaz
- Uno o más clientes que llaman a los métodos remotos

Librería MPI - Comunicadores

- Un comunicador define el dominio de comunicación.
- Cada proceso puede pertenecer a muchos comunicadores.
- Existe un comunicador que incluye a todos los procesos de la aplicación MPI_COMM_WORLD.
- Son variables del tipo MPI_Comm → almacena información sobre qué procesos pertenecen a él.
- En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar.

https://docs.google.com/document/d/1iULdohe9jO0AiYaAcbKrmb95LPBts-DyPG-O9yuQiPo/edit?usp=sharing

RPC(Remote Procedure Call) y Rendezvous (ADA)

Describa brevemente en qué consisten los mecanismos de RPC y Rendezvous. ¿Para qué tipo de problemas son más adecuados?

RPC solo provee un mecanismo de comunicación y la sincronización debe ser implementada por el programador utilizando algún método adicional. En cambio, Rendezvous es tanto un mecanismo de comunicación como de sincronización.
En RPC la comunicación se realiza mediante la ejecución de un CALL a un procedimiento, este llamado crea un nuevo proceso para ejecutar lo solicitado. El llamador se demora hasta que el proceso ejecute la operación requerida y le devuelva los resultados. En Rendezvous la diferencia con RPC está en cómo son atendidos los pedidos, el CALL es atendido por un proceso existente, no por uno nuevo. Es decir, con RPC el proceso que debe atender el pedido no se encuentra activo sino que se activa para responder al llamado y luego termina; en cambio, con Rendezvous el proceso que debe atender los requerimientos se encuentra continuamente activo. Esto hace que RPC permita servir varios pedidos al mismo tiempo y que con Rendezvous solo puedan ser atendidos de a uno por vez.
Estos mecanismos son más adecuados para problemas con interacción del tipo cliente servidor donde la comunicación entre ellos debe ser bidireccional y sincrónica, el cliente solicita un servicio y el servidor le responde con lo solicitado ya que el cliente no debe realizar ninguna otra tarea hasta no obtener una respuesta del servidor.

¿Por qué es necesario proveer sincronización dentro de los módulos RPC? ¿Cómo puede realizarse esta sincronización?

Es necesario proveer sincronización dentro de los módulos porque los procesos pueden ejecutarse concurrentemente. Estos mecanismos de sincronización pueden usarse tanto para el acceso a variables compartidas como para sincronizar interacciones entre los procesos si fuera necesario. En RPC existen dos modos de proveer sincronización y depende del modo en que se ejecutan los procesos dentro del módulo:
Si los procesos se ejecutan por exclusión mutua, sólo hay un proceso activo a la vez dentro del módulo, el acceso a las variables compartidas tiene exclusión mutua implícita pero la sincronización por condición debe programarse. Pueden usarse sentencias await o variables condición.
Si los procesos se ejecutan concurrentemente dentro del módulo debe implementarse tanto la exclusión mutua como la sincronización por condición para esto pueden usarse cualquiera de los mecanismos existentes como semáforos, monitores, PM o Rendezvous.

¿Qué elemento de la forma general de Rendezvous no se encuentra en ADA?

Rendezvous provee la posibilidad de asociar sentencias de scheduling y de poder usar los parámetros formales de la operación tanto en las sentencias de sincronización como en las sentencias de scheduling. Ada no provee estas posibilidades.

RPC y rendezvous son ideales para interacciones cliente/servidor. Ambas combinan aspectos de monitores y PMS. Como con monitores, un módulo o proceso exporta operaciones, y las operaciones son invocadas por una sentencia call. Como con las sentencias de salida en PMS, la ejecución de call demora al llamador. La novedad de RPC y rendezvous es que una operación es un canal de comunicación bidireccional desde el llamador al proceso que sirve el llamado y nuevamente hacia el llamador. En particular, el llamador se demora hasta que la operación llamada haya sido ejecutada y se devuelven 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. 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.

REMOTE PROCEDURE CALL
Con RPC, usaremos una componente de programa (el módulo) que contiene tanto procesos como procedures. 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. 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.

module Mname
  headers de procedures visibles
body
  declaraciones de variables
  código de inicialización
  cuerpos de procedures visibles
  procedures y procesos locales
end

RENDEZVOUS
Rendezvous combina las acciones de servir un llamado con otro procesamiento de la información transferida por el llamado. Con rendezvous, un proceso exporta operaciones que pueden ser llamadas por otros. Una declaración de proceso tendrá la siguiente forma:

pname:: declaraciones de operación
        declaraciones de variables
        sentencias

Las declaraciones especifican los headers de las operaciones servidas por el proceso.
Un proceso invoca una operación por medio de una sentencia call, la cual en este caso nombra otro proceso y una operación en ese proceso. Pero en contraste con RPC, una operación es servida por el proceso que la exporta. Por lo tanto, las operaciones son servidas una a la vez en lugar de concurrentemente. Si un proceso exporta una operación op, puede rendezvous con un llamador de op ejecutando:

in opname (parámetros formales) S; ni

Llamaremos a las partes entre las palabras claves operación guardada. La guarda nombra una operación y sus parámetros formales; el cuerpo contiene una lista de sentencias S. El alcance de los parámetros formales es la operación guardada entera.
En particular, in demora al proceso servidor hasta que haya al menos un llamado pendiente de op. Luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta S, y finalmente retorna los parámetros resultados al llamador. En ese punto, ambos procesos (el que ejecuta el in y el que llamó a op) pueden continuar la ejecución.

Para combinar comunicación guardada con rendezvous, generalizamos la sentencia in como sigue:

in op1 (formales1) and B1 by e1 S1
[] .....
[] opn (formalesn) and Bn by en Sn
ni

Cada operación guardada nuevamente nombra una operación y sus parámetros formales, y contiene una lista de sentencias. Sin embargo, la guarda también puede contener dos partes opcionales. La segunda parte es una expresión de sincronización (and Bi); si se omite, se asume que Bi es true. La tercera parte de una guarda es una expresión de scheduling (by ei); (El lenguaje Ada soporta rendezvous por medio de la sentencia accept y comunicación guardada por medio de la sentencia select.
El accept es como la forma básica de in, pero el select es menos poderoso que la forma general de in. Esto es porque select no puede referenciar argumentos a operaciones o contener expresiones de scheduling).
Una guarda en una operación guardada tiene éxito cuando

(1) la operación fue llamada, y
(2) la expresión de sincronización correspondiente es verdadera.

La ejecución de in se demora hasta que alguna guarda tenga éxito. Como es usual, si más de una guarda tiene éxito, una de ellas es elegida no deterministicamente. Si no hay expresión de scheduling, la sentencia in sirve la invocación más vieja que hace que la guarda tenga éxito.
Una expresión de scheduling se usa para alterar el orden de servicio de invocaciones por default (primero la invocación más vieja).

Mecanismos de comunicación y sincronización provistos por ADA

En ADA el rendezvous es el único mecanismo de sincronización y también es el mecanismo de comunicación primario.
Las declaraciones de entry tienen la forma entry identificador (formales). Los parámetros del entry pueden ser in, out o in out.
Ada también soporta arreglos de entries, llamados familias de entry.
Si la task T declara el entry E, otras tasks en el alcance de la especificación de T pueden invocar a E por una sentencia call: call T.E.(reales). Como es usual, la ejecución de call demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).
La task que declara un entry sirve a llamados de ese entry por medio de la sentencia accept. Esto tiene la forma general: accept E(formales) do lista de sentencias end;
La ejecución de accept demora la tarea hasta que haya una invocación de E, copia los argumentos de entrada en los formales de entrada y luego ejecuta la lista de sentencias. Cuando la lista de sentencias termina, los parámetros formales de salida son copiados a los argumentos de salida. En ese punto, tanto el llamador como el proceso ejecutante continúan.
Para controlar el no determinismo, Ada provee tres clases de sentencias select: wait selectivo, entry call condicional, y entry call timed.

La **sentencia wait selectiva** soporta comunicación guardada. La forma más común de esta sentencia es:

select when B1 sentencia **accept** E1; sentencias1
or ...
or when Bn sentencia **accept** En; sentenciasn
end select

Cada línea (salvo la última) se llama alternativa. Las Bi son expresiones booleanas, y las cláusulas when son opcionales. Una alternativa se dice que está abierta si Bi es true o se omite la cláusula when. Esta forma de wait selectivo demora al proceso ejecutante hasta que la sentencia accept en alguna alternativa abierta pueda ser ejecutada, es decir, haya una invocación pendiente del entry nombrado en la sentencia accept.

Dado que cada guarda Bi precede una sentencia accept, no puede referenciar los parámetros de un entry call. Además, Ada no provee expresiones de scheduling, lo cual hace difícil resolver algunos problemas de sincronización y scheduling.
La sentencia wait selectiva puede contener una alternativa opcional else, la cual es seleccionada si ninguna otra alternativa puede serlo.
En lugar de la sentencia accept, el programador puede también usar una sentencia delay o una alternativa terminate.
Una alternativa abierta con una sentencia delay es seleccionada si transcurrió el intervalo de delay; esto provee un mecanismo de timeout. La alternativa terminate es seleccionada esencialmente si todas las tasks terminaron o están esperando una alternativa terminate.
Estas distintas formas de sentencias wait selectiva proveen una gran flexibilidad, pero también resultan en un número algo confuso de distintas combinaciones. Para “empeorar” las cosas, hay dos clases adicionales de sentencia select.

Un **entry call condicional** se usa si una task quiere hacer polling de otra. El entry call es seleccionado si puede ser ejecutado inmediatamente; en otro caso, se selecciona la alternativa else. Tiene la forma:

select entry call; sentencias adicionales
else sentencias
end select

Un **entry call timed** se usa si una task llamadora quiere esperar a lo sumo un cierto intervalo de tiempo. En este caso, el entry call se selecciona si puede ser ejecutado antes de que expire el intervalo de delay. Esta sentencia soporta timeout, en este caso, de un call en lugar de un accept. Su forma es similar a la de un entry call condicional:

select entry call; sentencias
or sentencia de delay; sentencias
end select

Ada provee unos pocos mecanismos adicionales para programación concurrente. Las tasks pueden compartir variables; sin embargo, no pueden asumir que estas variables son actualizadas excepto en puntos de sincronización (por ej, sentencias de rendezvous). La sentencia abort permite que una tarea haga terminar a otra. Hay un mecanismo para setear la prioridad de una task. Finalmente, hay atributos que habilitan para determinar cuándo una task es llamable o ha terminado o para determinar el número de invocaciones pendientes de un entry.

Analice qué tipos de mecanismos de pasaje de mensajes son más adecuados para resolver problemas del tipo Cliente-Servidor, Pares que interactúan, filtro y productores-consumidores. Justificar.

- **Pares que interactúan, filtros y productor-consumidor:** Es más adecuado el uso PM ya que el flujo de comunicación es unidireccional. Además con RPC los procesos no pueden comunicarse directamente por lo que la comunicación debe ser implementada y con Rendezvous aunque los procesos si pueden comunicarse, no pueden ejecutar una llamada seguida de una entrada de datos por lo que deberían definirse procesos asimétricos o utilizar procesos helpers.
  Dependiendo del tipo de problema particular dependerá si es más adecuado PMA que provee buffering implícito y mayor grado de concurrencia o PMS que provee mayor sincronización.
- **Cliente-servidor:** Es más adecuado el uso de RPC o Rendezvous ya que el problema requiere de un flujo de comunicación bidireccional el cliente solicita un servicio y el servidor responde a la solicitud, es decir que no solo el cliente le envía información para llevar a cabo la operación sino que el servidor le debe devolver los resultados de dicha ejecución, también no es necesario que el cliente siga procesando ya que antes de realizar otra tarea requerirá los resultados por parte del servidor. Ambos mecanismos proveen este flujo de comunicación bidireccional, pero con PM deberían utilizarse dos canales uno para las solicitudes y otro para las respuestas.

Arquitectura Paralelas, Algoritmos Paralelos y Métricas

Clasificación de arquitecturas paralelas

Por la organización del espacio de direcciones.

- Memoria compartida: accesos a memoria comportado
    - Esquemas UMA
    - Esquemas NUMA
- Memoria distribuida: intercambio de mensajes

Por la granularidad.

Por el mecanismo de control.

- SISD
- SIMD
- MISD
- MIMD

Por la red de interconexión.

- Redes estaticas: constan de links punto a puto, tipicamente se usan para maquinas de pasaje de mensajes
- Redes dinámicas: están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de memoria compartida.

En qué consisten las arquitecturas SIMD y MIMD? ¿Para qué tipo de aplicaciones es más adecuada cada una? → Clasificación por el Mecanismo de Control

MIMD: cada procesador tiene su propio flujo de instrucciones y de datos entonces cada uno ejecuta su propio programa. Pueden ser con memoria compartida o distribuida. Logra paralelismo total, en cualquier momento se pueden estar ejecutando diferentes instrucciones con diferentes datos en procesadores diferentes. Ideal para simulaciones, comunicación y diseño

Sub-clasificación de MIMD:

- MPMD (multiple program multiple data): cada procesador ejecuta su propio programa (ejemplo con PVM).
- SPMD (single program multiple data): hay un único programa fuente y cada procesador ejecuta su copia independientemente (ejemplo con MPI).

SIMD: tiene múltiples flujos de datos pero sólo un flujo de instrucción. En particular, cada procesador ejecuta exactamente la misma secuencia de instrucciones, y lo hacen en un lockstep. Esto hace a las máquinas SIMD especialmente adecuadas para ejecutar algoritmos paralelos de datos. Logra un paralelismo a nivel de datos, o sea ejecuta la misma instrucción en todos los procesadores.

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 instrucciones (decodificadas por la UC) sobre los datos.

La memoria recibe y almacena datos en las escrituras, y brinda datos en las lecturas.

- Ejecución determinística.

MISD: Multiple Instruction Single Data

- Los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes.
- Operación sincrónica (en lockstep).
- No son máquinas de propósito general (“hipotéticas”, Duncan).
- Ejemplos posibles:
    - Múltiples filtros de frecuencia operando sobre una única señal.
    - Múltiples algoritmos de criptografía intentando crackear un único mensaje codificado.

Diseño de algoritmos Paralelos

Para diseñar un algoritmo paralelo se deben realizar alguno de los siguientes pasos:

- Identificar porciones de trabajo (tareas) concurrentes.
- Mapear tareas a procesos en distintos procesadores.
- Distribuir datos de entrada, intermedios y de salida.
- Manejo de acceso a datos compartidos.
- Sincronizar procesos.

Pasos Fundamentales: **Descomposición en Tareas** y **Mapeo de Procesos a Procesadores.**

Descomposición en Tareas

- Para desarrollar un algoritmo paralelo el primer punto es descomponer el problema en sus componentes funcionales concurrentes(procesos/tareas).
- Se trata de definir un gran número de pequeñas tareas para obtener una descomposición de grano fino, para brindar la mayor flexibilidad a los algoritmos paralelos potenciales.
- En etapas posteriores, la evaluación de los requerimientos de comunicación, arquitectura de destino, o temas de IS (Ingeniería de Software) pueden llevar a descartar algunas posibilidades detectadas en esta etapa, revisando la partición original y aglomerando tareas para incrementar su tamaño o granularidad.
- Esta descomposición puede realizarse de muchos modos. Un primer concepto es pensar en tareas de igual código (normalmente **paralelismo de datos o dominio**) pero también podemos tener diferente código **(paralelismo funcional**).

- **Descomposición de datos:** determinar una división de los datos (en muchos casos, de igual tamaño) y luego asociar el cómputo (típicamente, cada operación con los datos con que opera).
    - Esto da un número de tareas, donde cada uno comprende algunos datos y un conjunto de operaciones sobre ellos. Una operación puede requerir datos de varias tareas, y esto llevará a la **comunicación**.
    - Son posibles distintas particiones, basadas en diferentes estructuras de datos. Por ejemplo, diferentes formas de descomponer una estructura 3D de datos. Inicialmente la de grano más fino.

- **Descomposición funcional:** primero descompone el cómputo en tareas disjuntas y luego trata los datos.
    - La descomposición funcional tiene un rol importante como técnica de estructuración del programa, para reducir la complejidad del diseño general. Modelos computacionales de sistemas complejos pueden estructurarse como conjuntos de modelos más simples conectados por interfaces

Aglomeración

- El algoritmo resultante de las etapas anteriores es abstracto en el sentido de que no es especializado para ejecución eficiente en una máquina particular.
- Esta etapa revisa las decisiones tomadas con la visión de obtener un algoritmo que ejecute en forma eficiente en una clase de máquina real.
- **En particular, se considera si es útil combinar o aglomerar las tareas para obtener otras de mayor tamaño. También se define si vale la pena replicar datos y/o computación.**

3 objetivos, a veces conflictivos, que guían las decisiones de aglomeración y replicación:

- **Incremento de la granularidad:** intenta reducir la cantidad de comunicaciones combinando varias tareas relacionadas en una sola.
- **Preservación de la flexibilidad:** al juntar tareas puede limitarse la escalabilidad del algoritmo. La capacidad para crear un número variante de tareas es crítica si se busca un programa portable y escalable.
- **Reducción de costos de IS (Ingeniería de Software):** se intenta evitar cambios extensivos, por ejemplo, reutilizando rutinas existentes.

Características de las tareas

Una vez que tenemos el problema separado en tareas conceptualmente independientes, tenemos una serie de características de las mismas que impactarán en la perfomance alcanzable por el algoritmo paralelo:

- Generación de las tareas.
- El tamaño de las tareas.
- Conocimiento del tamaño de las tareas.
- El volumen de datos asociado con cada tarea

Mapeo de procesos a procesadores

- Se especifica dónde ejecuta cada tarea.
- Este problema no existe en uniprocesadores o máquinas de memoria compartida con scheduling de tareas automático.
- Objetivo: minimizar tiempo de ejecución. Dos estrategias, que a veces conflictúan: ubicar tareas que pueden ejecutar concurrentemente en distintos procesadores para mejorar la concurrencia o poner tareas que se comunican con frecuencia en iguales procesadores para incrementar la localidad.
- El problema es NP-completo: no existe un algoritmo de tiempo polinomial tratable computacionalmente para evaluar tradeoffs entre estrategias en el caso general. Existen heurísticas para clases de problema.
- Normalmente tendremos más tareas que procesadores físicos.
- Los algoritmos paralelos (o el scheduler de ejecución) deben proveer un mecanismo de “mapping” entre tareas y procesadores físicos.
- Nuestro lenguaje de especificación de algoritmos paralelos debe poder indicar claramente las tareas que pueden ejecutarse concurrentemente y su precedencia/prioridad para el caso que no haya suficientes procesadores para atenderlas.
- La dependencia de tareas condicionarán el balance de carga entre procesadores.
- La interacción entre tareas debe tender a minimizar la comunicación de datos entre procesadores físicos.

Criterio para el mapeo de tareas a procesadores

Un buen mapping es crítico para el rendimiento de los algoritmos paralelos.

1. Tratar de mapear tareas independientes a diferentes procesadores.
2. Asignar prioritariamente los procesadores disponibles a las tareas que estén en el camino crítico.
3. Asignar tareas con alto nivel de interacción al mismo procesador, de modo de disminuir el tiempo de comunicación físico.

- Notar que estos criterios pueden oponerse entre sí … por ejemplo el criterio 3 puede llevarnos a NO paralelizar.
- Debe encontrarse un equilibrio que optimice el rendimiento paralelo ⇒ MAPPING DETERMINA LA EFICIENCIA DEL ALGORITMO.

MÉTRICAS

Defina las métricas de speedup y eficiencia. Cuál es el significado de cada una de ellas (qué miden)? Ejemplifique. En qué consiste la “ley de Amdahl”.

Ambas técnicas son métricas relativas que representan la fracción de tiempo que los procesadores emplean realizando tareas útiles para la resolución de los algoritmos. Estas métricas caracterizan la efectividad con la que el algoritmo paralelo usa los recursos de las computadoras.

- **Speedup**⇒ S = Ts/ Tp: S es el cociente entre el tiempo de ejecución secuencial del algoritmo secuencial conocido más rápido (Ts) y el tiempo de ejecución paralelo del algoritmo elegido (Tp). El rango de valores de S va desde 0 a p, siendo p el número de procesadores. Mide cuánto más rápido es el algoritmo paralelo con respecto al algoritmo secuencial, es decir, cuánto se gana por usar más procesadores.

- **Eficiencia⇒** E = S/P: Cociente entre speedup y número de procesadores. El valor está entre 0 y 1, dependiendo de la efectividad en el uso de los procesadores. Cuando es 1 corresponde al speedup perfecto. Mide la fracción de tiempo en que los procesadores son útiles para el cómputo, es decir cuánto estoy usando de los recursos disponibles.

En cualquier programa paralelizado existen dos tipos de código; el código paralelizado y el código secuencial. Existen ciertas secciones de código que ya sea por dependencias, por acceso a recursos únicos o por requerimientos del problema no pueden ser paralelizadas. Estas secciones conforman el código secuencial, que debe ser ejecutado por un solo elemento del procesador.
Entonces, es lógico afirmar que la mejora del speedup de un programa dependerá del tiempo en el que se ejecuta el código secuencial, el tiempo en el que se ejecuta el código paralelizable y el número de operaciones ejecutadas de forma paralela.

La **"Ley de Amdahl"** enuncia que para cualquier tipo de problema existe un máximo speedup alcanzable que no depende de la cantidad de procesadores que se utilicen para resolverlo. Esto es así porque llega un momento en que no existe manera de aumentar el paralelismo de un programa.

¿Cuál es el objetivo de la programación paralela?

El objetivo principal de la programación paralela es reducir el tiempo de ejecución o resolver problemas más grandes o con mayor precisión en el mismo tiempo. Al contrario que en la programación concurrente esta técnica enfatiza la verdadera simultaneidad en el tiempo de la ejecución de las tareas.

Mencione al menos 4 problemas en los cuales Ud. Entiende que es conveniente el uso de técnicas de programación paralela.

- Procesamiento de imágenes
- Simulación de circulación oceánica.
- Multiplicación de matrices
- Generación de números primos

Ejercicio Maldito

5. (2 puntos) Métricas en Sistemas Paralelos: Sea la siguiente solución a un problema de matrices de nxn con P procesos en paralelo con variables compartidas. Suponga n = 320 y cada procesador capaz de ejecutar un proceso. NOTA: para hacer los cálculos sólo tenga en cuenta las operaciones realizadas en la instrucción dentro del for Z.

```
int a[n,n], b[n,n], c[n,n], d[n,n]; --Ya inicializadas

process worker [w: 0..P-1] {
  int primera = w*(n/P) + 1;
  int ultima = primera + (n/P) -1;
  for (x = primera; x <= ultima; x++)
    { for (y = 0; y < n; y++)
        { c[x,y] = 0;
          for (z = 0; z < n; z++)
              c[x,y] = c[x,y] + (a[x,z]*b[z,y]) + d[y,z];
        }
    }
}
```

a. Calcular cuántas asignaciones, sumas y productos se hacen secuencialmente (caso en que P=1); y cuántas se realizan en cada procesador en la solución paralela con P=8.

N = 320

Análisis Secuencial con P=1

Asignaciones: $320^3 = 32.768.000$
Sumas: $320^3 + 320^3 = 65.536.000$
Productos: $320^3 = 32.768.000$

Análisis Paralelo con P=8 (320/8 = 40)

Asignaciones: $320^2 * 40 = 4.096.000$
Sumas: $320^2 * 40 + 320^2 * 40 = 8.192.000$
Productos: $320^2 * 40 = 4.096.00$

b. Dados que los procesadores de P1 y P2 son idénticos, con tiempos de asignación 2, de suma 4 y de producto 6; los procesadores P3 y P4 son el doble de potentes (tiempos 1, 2 y 3 para asignaciones, sumas y productos respectivamente); y el resto de los procesadores (de P5 a P8) son la mitad de potente que P1 (tiempos 4, 8 y 12 para asignaciones, sumas y productos respectivamente). Calcular cuánto tarda el programa paralelo y el secuencial.

En estos casos tomamos los valores del análisis paralelo para resolver

P1 y P2
Asignaciones * 2 → 4.096.000 * 2 = 8.192.000
Sumar * 4 → 8.192.000 * 4 = 32.768.000
Producto * 6 → 4.096.000 * 6 = 24.576.000

P3 y P4
Asignar * 1 → 4.096.000 * 1 = 4.096.000
Sumar * 2 → 8.192.000 * 2 = 16.384.000
Producto * 3 → 4.096.000 * 3 = 12.288.000

P5 a P8
Asignar * 4 → 4.096.000 * 4 = 16.384.000
Sumar * 8 → 8.192.000 * 8 = 65.536.000
Producto * 12 → 4.096.000 * 12 = 49.152.000

Cuanto tarda paralelo:
Para esto tomamos el tiempo paralelo calculado en el paso 1 y lo multiplicamos por el tiempo del procesador más lento, en este caso sería lo mismo que el cálculo realizado con P5 a P8. En este caso, hay que sumar las unidades de tiempo

16.384.000 + 65.536.000 + 49.152.000 = **131.072.000**

Cuanto tarda secuencial:
Para esto tomamos el tiempo secuencial calculado en el paso 1 y lo multiplicamos por el tiempo del procesador más rápido (serían P3 y P4). En este caso, hay que sumar las unidades de tiempo

32.768.000 * 1 = 32.768.000
65.536.000 * 2 = 131.072.000
32.768.000 * 3 = 98.304.000

32.768.000 + 131.072.000 + 98.304.000 = **262.144.000**

c. Realizar paso a paso el cálculo del valor del speedup y la eficiencia.

Speedup: TS/TP
262.144.000 / 131.072.000 = **2**

Eficiencia: En este caso es una arquitectura de procesadores heterogéneos por lo que deberíamos calcular el Speedup Óptimo para calcular la Eficiencia. Sabemos por el enunciado que P3 y P4 son el doble de potentes que P1 y P2, y que P5 a P8 son la mitad de potentes que P1.
Tomamos como potencia relativa la más rápida, en este caso la de P3 y P4, igualándola a 1.
La potencia de P1 y P2, sería la potencia relativa dividido 2 → 1 / 2 = 0,5
Y la potencia de P5 a P8, sería el resultado de P1 dividido 2 → 0,5 / 2 = 0,25

P1 = 0,5/1 = 0,5
P2 = 0,5/1 = 0,5
P3 = 1/1 = 1
P4 = 1/1 = 1
P5 = 0,25/1 = 0,25
P6 = 0,25/1 = 0,25
P7 = 0,25/1 = 0,25
P8 = 0,25/1 = 0,25

Se suman todos dando un Total de = 4
Speedup / Speedup Óptimo → 2/4 = **0,5**
Eficiencia: 0,5

Si fueran procesadores homogéneos sería Speedup / Procesadores = 2/8 = 0.25

d. Modificar el código para lograr una mejor eficiencia. Calcular la nueva eficiencia y comparar con la calculada en (c).

Para mejorar la eficiencia hay que hacer un mejor balance de carga, dándole a los procesadores más lentos menor cantidad de strips.
Antes todos recibían 40 strips

P1 = 40	repartir nuevamente	P1 = 40
P2 = 40		P2 = 40
P3 = 40		P3 = 80
P4 = 40		P4 = 80
P5 = 40		P5 = 20
P6 = 40		P6 = 20
P7 = 40		P7 = 20
P8 = 40		P8 = 20

SpeedUp = TS/TP = 262.144.000 / 65.536.000 = 4
Speedup / Speedup Óptimo → 4/4 = 1

Perdón por hacernos esto.