Resumen Programación Concurrente

Clase 1

Concurrencia

¿Qué es?

- Concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente
- Permite a distintos objetos actuar al mismo tiempo
- Factor relevante para el diseño de hardware, sistemas operativos, multiprocesadores, computación distribuida, programación y diseño

¿Donde está?

- Navegador Web accediendo una página mientras atiende al usuario.
- Varios navegadores accediendo a la misma página
- Acceso a disco mientras otras aplicaciones siguen funcionando
- Impresión de un documento mientras se consulta
- Teléfono avisa recepción de llamada mientras se habla
- Cualquier objeto más o menos "inteligente" exhibe concurrencia
- Juegos, automóviles, etc.

Los sistemas biológicos suelen ser masivamente concurrentes: comprenden un gran número de células, evolucionando simultáneamente y realizando (independientemente) sus procesos. En el mundo biológico los sistemas secuenciales rara vez se encuentran.
En algunos casos se tiende a pensar en sistemas secuenciales en lugar de concurrentes para simplificar el proceso de diseño. Pero esto va en contra de la necesidad de sistemas de cómputo cada vez más poderosos y flexibles.

Concurrencia "natural"

- Obliga a establecer un orden en el despliegue de cada cartel.
- Código más complejo de desarrollar y mantener
- ¿Qué pasa si se tienen más de dos carteles?
- **Más natural**: cada cartel es un elemento independiente que actúa concurrentemente con otros --> **es decir, ejecutar dos o más algoritmos simples concurrentemente**

```
Programa Cartel (color, tiempo)
    Mientras (true)
        Demorar (tiempo segundos)
```

```
        Desplegar cartel (color)
    Fin mientras
Fin programa
```

- No hay un orden preestablecido en la ejecución ⇒ no determinismo (ejecuciones con la misma "entrada" puede generar diferentes "salidas")

¿Por qué es necesaria la Programación Concurrente?

- No hay más ciclos de reloj -> Multicore -> ¿por qué? y ¿para qué?
- Aplicaciones con estructura más natural.
  - El mundo no es secuencial.
  - Más apropiado programar múltiples actividades independientes y concurrentes.
  - Reacción a entradas asincrónicas (ej: sensores en un STR).
- Mejora en la respuesta
  - No bloquear la aplicación completa por E/S.
  - Incremento en el rendimiento de la aplicación por mejor uso del hardware (ejecución paralela).
- Sistemas distribuidos
  - Una aplicación en varias máquinas.
  - Sistemas C/S o P2P.

Objetivo de los sistemas concurrentes

> **Important**
>
> Ajustar el modelo de arquitectura del hardware y software al problema del mundo real a resolver.

> **Important**
>
> Incrementar la performance, mejorando los tiempos de respuesta de los sistemas de cómputo, a través de un enfoque diferente de la arquitectura física y lógica de las soluciones

Algunas ventajas

- La velocidad de ejecución que se puede alcanzar.
- Mejor utilización de la CPU de cada procesador.
- Explotación de la concurrencia inherente a la mayoría de los problemas reales.

Posible comportamiento de los procesos

Programa Secuencial:

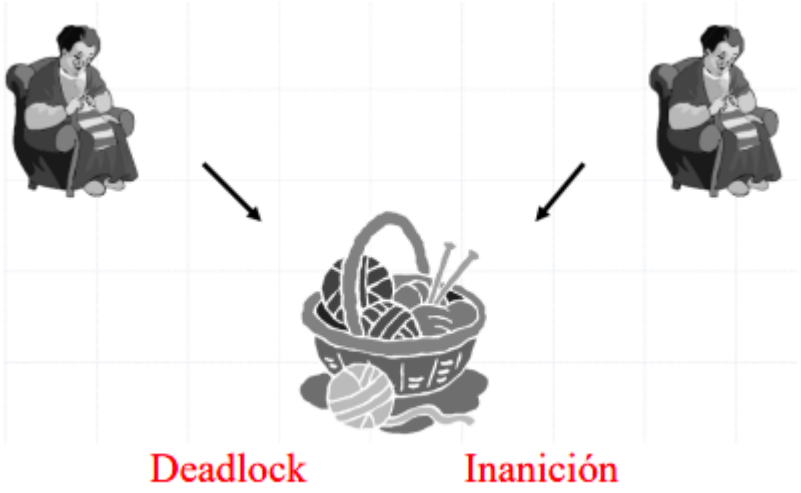
Un único flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente.
Por ahora llamaremos “Proceso” a un programa secuencial.
Un único hilo o flujo de control
→ programación secuencial, monoprocesador.
Múltiples hilos o flujos de control
→ programa concurrente.
→ programa paralelos.
Los procesos cooperan y compiten...

Procesos independientes

- Relativamente raros.
- Poco interesante.

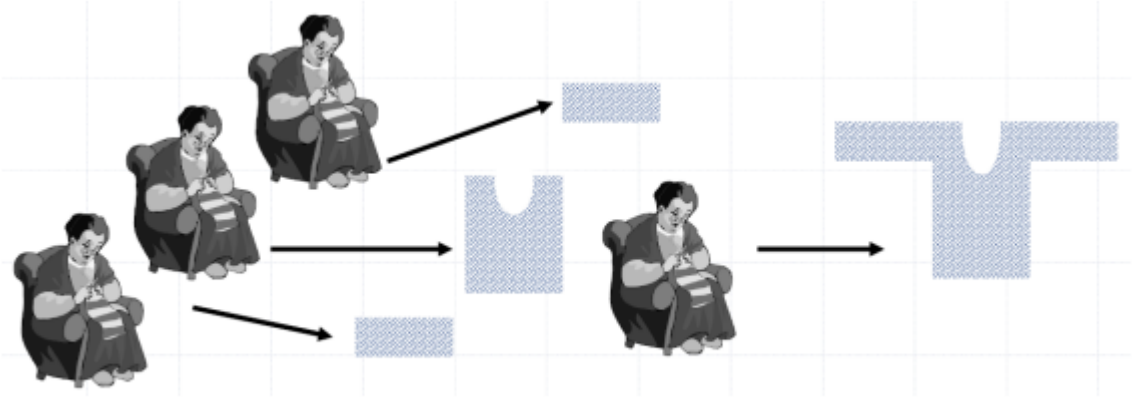
Competencia

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



Cooperación

- Los procesos se combinan para resolver una tarea común.
- Sincronización.



Procesamiento secuencial, concurrente y paralelo

Analicemos la solución secuencial y monoprocesador (una máquina) para fabricar un objeto compuesto por N partes o módulos.
La solución secuencial nos fuerza a establecer un estricto orden temporal.
Al disponer de sólo una máquina, el ensamblado final del objeto se podrá realizar luego de N pasos de procesamiento (la fabricación de cada parte).

Si disponemos de N máquinas para fabricar el objeto, y no hay dependencia (por ejemplo de la materia prima), cada una puede trabajar al mismo tiempo en una parte. Solución Paralela.
Consecuencias ⇒

- Menor tiempo para completar el trabajo.
- Menor esfuerzo individual.
- Paralelismo del hardware.
  Dificultades ⇒
- Distribución de la carga de trabajo (diferente tamaño o tiempo de fabricación de cada parte, diferentes especializaciones de cada máquina y/o velocidades).
- Necesidad de compartir recursos evitando conflictos.
- Necesidad de esperarse en puntos clave.
- Necesidad de comunicarse.
- Tratamiento de las fallas.
- Asignación de una de las máquinas para el ensamblado (¿Cual?)

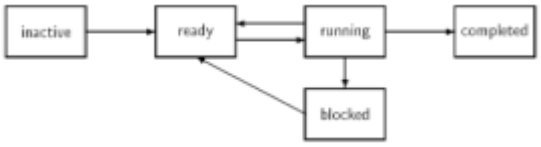
Otro enfoque: un sóla máquina dedica una parte del tiempo a cada componente del objeto ⇒ Concurrencia sin paralelismo de hardware ⇒Menor speedup.
Dificultades ⇒

- Distribución de carga de trabajo.
- Necesidad de compartir recursos.
- Necesidad de esperarse en puntos clave.
- Necesidad de comunicarse.
- Necesidad de recuperar el “estado” de cada proceso al retomarlo.
  CONCURRENCIA ⇒Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.

Este último caso sería multiprogramación en un procesador

- El tiempo de CPU es compartido entre varios procesos, por ejemplo por time slicing.
- El sistema operativo controla y planifica procesos: si el slice expiró o el proceso se bloquea el sistema operativo hace context (process) switch.
  - Process switch: suspender el proceso actual y restaurar otro
    1. Salvar el estado actual en memoria. Agregar el proceso al final de la cola de ready o una cola de wait.
    2. Sacar un proceso de la cabeza de la cola ready. Restaurar su estado y ponerlo a correr.
  - Reanudar un proceso bloqueado: mover un proceso de la cola de wait a la de ready

- Estado de los procesos



Programa concurrente

> **Definicion**
>
> Un programa concurrente especifica dos o más "programas secuenciales" que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos

Un proceso o tarea es un elemento concurrente abstracto que puede ejecutarse simultáneamente con otros procesos o tareas, si el hardware lo permite (por ejemplo los TASKs de ADA).
Un programa concurrente puede tener N procesos habilitados para ejecutarse concurrentemente y un sistema concurrente puede disponer de M procesadores cada uno de los cuales puede ejecutar uno o más procesos.
Características importantes:

- interacción
- no determinismo ⇒ dificultad para la interpretación y debug

Procesos e hilos

Procesos: Cada proceso tiene su propio espacio de direcciones y recursos

Procesos livianos, threads o hilos:

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

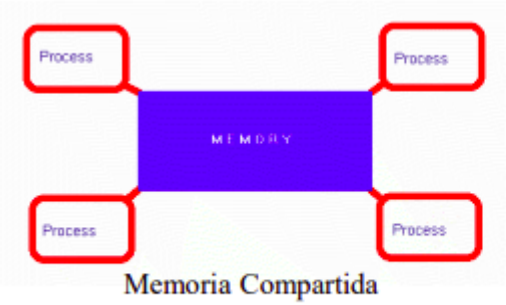
Conceptos básicos de concurrencia

Comunicación entre procesos

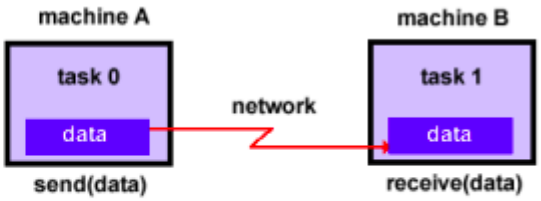
La comunicación entre procesos concurrentes indica el modo en que se organizan y trasmiten datos entre tareas concurrentes. Esta organización requiere especificar protocolos para controlar el progreso y la corrección. Los procesos se COMUNICAN:

- Por Memoria Compartida.

- Por Pasaje de Mensajes.



Memoria Compartida



Pasaje de Mensajes

Memoria Compartida

- Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella.
- Lógicamente no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a bloquear y liberar el acceso a la memoria.
- La solución más elemental es una variable de control tipo “semáforo” que habilite o no el acceso de un proceso a la memoria compartida.

Pasaje de mensajes

- Es necesario establecer un canal (lógico o físico) para transmitir información entre procesos.
- También el lenguaje debe proveer un protocolo adecuado.
- Para que la comunicación sea efectiva los procesos deben “saber” cuándo tienen mensajes para leer y cuando deben trasmitir mensajes.

Sincronización entre procesos

Definicion

La sincronización es la posesión de información acerca de otro proceso para coordinar actividades. Los procesos se sincronizan:

- Por exclusión mutua.
- Por condición.

Sincronización por exclusión mutua

- Asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
- Si el programa tiene secciones críticas que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.

Sincronización por condición

- Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

Ejemplo de los dos mecanismos de sincronización en un problema de utilización de un área de memoria compartida (buffer limitado con productores y consumidores).

Interferencia

> **Definicion**
>
> Un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

Ejemplo 1: nunca se debería dividir por 0.

```
int x, y, z;
process A1
{ ….
    y = 0;
    ….
}
process A2
{ …..
    if (y <> 0) z = x/y;
    ..…
}
```

Ejemplo 2: siempre público debería terminar con valor igual a E1+E2

```
int Público = 0
process B1
{ int E1 = 0;
  for i= 1..100
    { esperar llegada
        E1 = E1 + 1;
        Público = Público + 1;
    }
}
process B2
{ int E2 = 0;
  for i= 1..100
   { esperar llegada
     E2 = E2 + 1;
     Público = Público + 1;
    }
}
```

Prioridad y granularidad

> **Important**
>
> Un proceso que tiene mayor prioridad puede causar la suspensión (preemption) de otro proceso concurrente

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

Manejo de los recursos

Uno de los temas principales de la programación concurrente es la administración de recursos compartidos:

- Esto incluye la asignación de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.
- Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (fairness).
- Dos situaciones NO deseadas en los programas concurrentes son la inanición de un proceso (no logra acceder a los recursos compartidos) y el overloading de un proceso (la carga asignada excede su capacidad de procesamiento).
- Otro problema importante que se debe evitar es el deadlock.

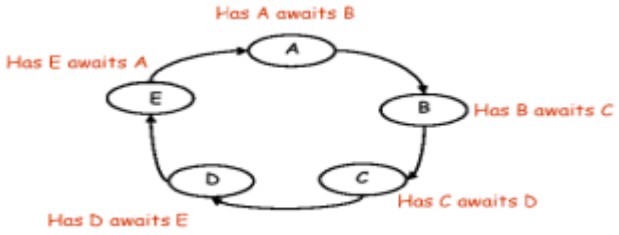
Problema de deadlock

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

4 propiedades necesarias y suficientes para que exista deadlock son:

- Recursos reusables serialmente: los procesos comparten recursos que pueden usar con exclusión mutua.
- Adquisición incremental: los procesos mantienen los recursos que poseen mientras esperar adquirir recursos adicionales.

- No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.
- Espera cíclica: existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.



Requerimientos para un lenguaje concurrente

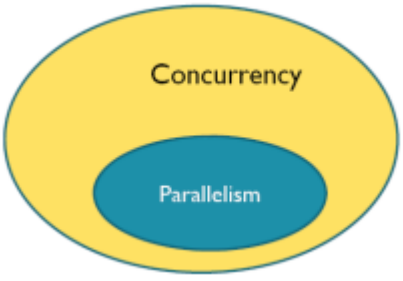
Independientemente del mecanismo de comunicación / sincronización entre procesos, los l**enguajes de programación concurrente** deberán proveer primitivas adecuadas para la especificación e implementación de las mismas.
Requerimientos de un lenguaje de programación concurrente:

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

Concurrencia y paralelismo

CONCURRENCIA ⇒ Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores. Especificar la concurrencia implica especificar los procesos concurrentes, su comunicación y su sincronización.

PARALELISMO ⇒ Se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.



Problemas asociados con la Programación concurrente

- Los procesos no son independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas.
- Los procesos iniciados dentro de un programa concurrente pueden NO estar "vivos". Esta pérdida de la propiedad de liveness puede indicar deadlocks o una mala distribución de recursos.
- Hay un no determinismo implícito en el interleaving de procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas ⇒

dificultad para la interpretación y debug.
- Posible reducción de performance por overhead de context switch, comunicación, sincronización, ...
- Mayor tiempo de desarrollo y puesta a punto. Difícil paralelizar algoritmos secuenciales.
- Necesidad de adaptar el software concurrente al hardware paralelo para mejora real en el rendimiento.

> **Definicion segun el libro**
>
> Una **propiedad de vivacidad** (**liveness**) es aquella en la que **el programa eventualmente entra en un estado bueno**, es decir, un estado en el que las variables tienen valores deseables o se cumple una condición esperada.

Concurrencia a nivel de hardware

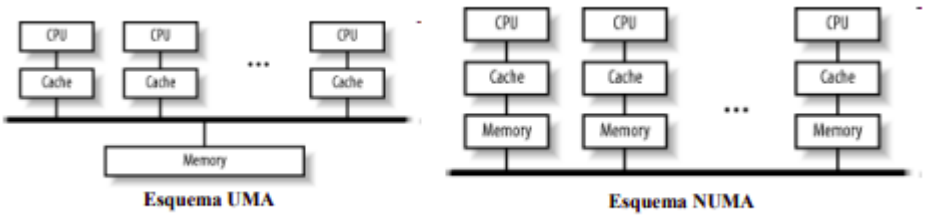
Límite físico en la velocidad de los procesadores

- Máquinas monoprocesador ya no pueden mejorar
- Más procesadores por chip para mayor potencia de cómputo
- Multicores -> Cluster multicores -> Consumo
- Uso eficiente -> Programación concurrente y paralela
  Niveles de memoria.
- Jerarquía de memoria. ¿Consistencia?
- Diferencias de tamaño y tiempo de acceso.
- Localidad temporal y espacial de los datos.

Máquinas de memoria compartida vs memoria distribuida.

Multiprocesadores de memoria compartida

- Interacción modificando datos en la memoria compartida
- Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos).
  Problemas de sincronización y consistencia
- Esquemas NUMA para mayor número de procesadores distribuidos
- Problema de consistencia



Multiprocesadores con memoria distribuida

- Procesadores conectados por una red.
- Memoria local (no hay problemas de consistencia).

- Interacción es sólo por pasaje de mensajes.
- Grado de acoplamiento de los procesadores:
  - Multicomputadores (máquinas fuertemente acopladas). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.
  - Memoria compartida distribuida.
  - Clusters.
  - Redes (multiprocesador débilmente acoplado).

Clases de Instrucciones

Programación secuencial y concurrente

Un programa concurrente esta formado por un conjunto de programas secuenciales.

- La programación secuencial estructurada puede expresarse con 3 clases de instrucciones básicas: asignación, alternativa ==(decisión) e ==iteración (repetición con condición).
- Se requiere una clase de instrucción para representar la concurrencia.

Declaraciones de variables

- Variable simple:
  `tipo variable = valor. Ej: int x= 8; int z,y;`
- Arreglos:

```
int a|10|; int e|3:10|
int b|10| = (|10|2)
int aa|5,5|;int ee|3:10,2:9|
int bb|5,5, = (|5|(|5|2))
```

Asignación

- Asignación simple: `x = e`
- Sentencia de asignación compuesta: `x = x + 1; y = y - 1; z = x + y; a|3| = 6; aa|2,5| = a|4|`
- Llamado a funciones: `x = f(y) + g(6) - 7`
- swap: v1:=: v2
- skip: termina inmediatamente y no tiene efecto sobre ninguna variable de programa

Alternativa

- Sentencias de alternativa simple:
  if B → S
  B expresión booleana. S instrucción simple o compuesta ({}).
  B “guarda” a S pues S no se ejecuta si B no es verdadera.
- Sentencias de alternativa múltiple:
  if B1 → S1

□ B2 → S2
.....
□ Bn → Sn
fi
Las guardas se evalúan en algún orden arbitrario.
Elección no determinística.
Si ninguna guarda es verdadera el if no tiene efecto.

- Otra opción:
  if (cond) S;
  if (cond) S1 else S2;

Ejemplo:

```
if p > 2 → p = p * 2
   □ p < 2 → p = p * 3
   □ p == 2 → p = 5
fi
```

Iteración

- Sentencias de alternativa ITERATIVA múltiple

```
do B1 → S1
    □ B2 → S2
    .....
    □ Bn → Sn
od
```

  Las sentencias guardadas son evaluadas y ejecutadas hasta que todas las guardas sean falsas
  La elección es no determinística si más de una guarda es verdadera
- For-all: forma genera de repetición e iteración
  `fa cuantificadores → Secuencia de Instrucciones af`
  Cuantificador ≡ variable := exp_inicial to exp_final st B
  El cuerpo del fa se ejecuta 1 vez por cada combinación de valores de las variables de iteración. Si hay cláusula such-that (st), la variable de iteración toma sólo los valores para los que *B* es true.
  Ejemplo: `fa i := 1 to n, j := i+1 to n st a[i] > a[j] → a[i] :=: a[j] af`
- Otra opción:
  `while (cond) S;` for [i = 1 to n, j = 1 to n st (j mod 2 = 0)] S;

Ejemplos:

```
do p > 0 → p = p - 2
    □ p < 0 → p = p + 3
    □ p == 0 → p = random(x)
od
```

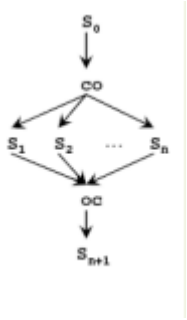
Inicialización de un vector

```
fa i := 1 to n → a[i] = 0 af
```

Ordenación de un vector de menor a mayor

```
fa i := 1 to n, j := i+1 to n st a[i] > a[j] → a[i] :=: a[j] af
```

Concurrencia

- **Sentencia co**:
  `co S1 // ..... // Sn oc` → ejecuta las Si tareas concurrentemente
  La ejecución del co termina cuando todas las tareas terminaron
  

  Cuantificadores
  `co [i=1 to n] { a[i]=0; b[i]=0 } oc` → Crea n tareas concurrentes.
- **Process:** otra forma de representar concurrencia
  `process A {sentencias}` → proceso único independiente.
  Cuantificadores.
  `process B [i=1 to n] {sentencias}` → n procesos independientes.
- **Diferencia:** process ejecuta en background, mientras el código que contiene un co espera a que el proceso creado por la sentencia co termine antes de ejecutar la sentencia

Acciones atómicas y sincronización

Atomicidad de grano fino

- **Estado** de un programa concurrente.
- Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
- Una **acción atómica** hace una transformación de estado indivisibles (estados intermedios invisibles para otros procesos).
- Ejecución de un programa concurrente → intercalado (interleaving) de las acciones atómicas ejecutadas por procesos individuales.
- **Historia** de un programa concurrente (trace): ejecución de un programa concurrente con un interleaving particular. En general el número de posibles historias de un programa concurrente es enorme; pero no todas son válidas.
- **Interacción** → determina cuales historias son correctas.
- Algunas historias son válidas y otras no.

- Se debe asegurar un orden temporal entre las acciones que ejecutan los procesos → las tareas se intercalan ⇒ deben fijarse restricciones.

La sincronización por condición permite restringir las historias de un programa concurrente para asegurar el orden temporal necesario.

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

no podemos confiar en la intuición para analizar un programa concurrente…

- En la mayoría de los sistemas el tiempo absoluto no es importante.
- Con frecuencia los sistemas son actualizados con componentes más rápidas. La corrección no debe depender del tiempo absoluto.
- El tiempo se ignora, sólo las secuencias son importantes
- Puede haber distintos ordenes (interleavings) en que se ejecutan las instrucciones de los diferentes procesos; los programas deben ser correctos para todos ellos.

En lo que sigue, supondremos máquinas con las siguientes características:

- Los valores de los tipos básicos se almacenan en elementos de memoria leídos y escritos como acciones atómicas.
- Los valores se cargan en registros, se opera sobre ellos, y luego se almacenan los resultados en memoria.
- Cada proceso tiene su propio conjunto de registros (context switching).
- Todo resultado intermedio de evaluar una expresión compleja se almacena en registros o en memoria privada del proceso.

- Si una expresión e en un proceso no referencia una variable alterada por otro proceso, la evaluación será atómica, aunque requiera ejecutar varias acciones atómicas de grano fino.
- Si una asignación $x = e$ en un proceso no referencia ninguna variable alterada por otro proceso, la ejecución de la asignación será atómica.
  Normalmente los programas concurrentes no son disjuntos ⇒ es necesario establecer algún requerimiento más débil …
  Referencia crítica en una expresión ⇒ referencia a una variable que es modificada por otro proceso. Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

Propiedad de "A lo sumo una vez"

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

1. e contiene a lo sumo una referencia crítica y x no es referenciada por otro proceso, o
2. e no contiene referencias críticas, en cuyo caso x puede ser leída por otro proceso.
   Una expresiones e que no está en una sentencia de asignación satisface la propiedad de “A lo sumo una vez” si no contiene más de una referencia crítica.

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

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

Ejemplos:

- `int x=0, y=0` ; 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`

Especificación de la sincronización

- Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente.
- En general, es necesario ejecutar secuencias de sentencias como una única acción atómica (sincronización por exclusión mutua) .

Mecanismo de sincronización para construir una acción atómica de grano grueso (coarse grained) como secuencia de acciones atómicas de grano fino (fine grained) que aparecen como indivisibles.
$\langle e \rangle$ indica que la expresión e debe ser evaluada atómicamente.
$\langle await(B)S; \rangle$ se utiliza para especificar sincronización
La expresió booleana B especifica una condición de demora.
S es una secuencia de sentencias que se garantiza que termina
Se garantiza que B es true cuando comienza la ejecución de S
Ningun estado interno de S es visible para los otros procesos

Sentencia con alto poder expresivo, pero el costo de implementación de la forma general de await (exclusión mutua y sincronización por condición) es alto.

- Await general: $\langle await(s > B)s = s - 1; \rangle$
- Await para exclusión mutua `⟨x = x + 1; y = y + 1⟩`
- Ejemplo await para sincronización por condición: `⟨await (count > 0)⟩`
  Si B satisface ASV, puede implementarse como busy waiting o spin loop
  do (not B) → skip od (while (not B);)

Acciones atómicas incondicionales y condicionales

Ejemplo: productor/consumidor con buffer de tamaño N

```
cant: int = 0;
Buffer: cola;
process Productor
{ while (true)
    Generar Elemento
    <await (cant < N); push(buffer, elemento); cant++ >
}
process Consumidor
{ while (true) 0)
    <await (cant > 0); pop(buffer, elemento); cant-- >
    Consumir Elemento
}
```

Propiedades y Fairness

Propiedad de seguridad y vida

Una propiedad de un programa concurrente es un atributo verdadero en cualquiera de las historias de ejecución del mismo Toda propiedad puede ser formulada en términos de dos clases: seguridad y vida. 73

- Seguridad (safety)
  - Nada malo le ocurre a un proceso: asegura estados consistentes.
  - Una falla de seguridad indica que algo anda mal.
  - Ejemplos de propiedades de seguridad: exclusión mutua, ausencia de interferencia entre procesos, partial correctness.
- Vida (liveness)
  - Eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks.
  - Una falla de vida indica que las cosas dejan de ejecutar.
  - Ejemplos de vida: terminación, asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, etc ⇒ dependen de las políticas de scheduling.

Fairness y políticas de scheduling

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

Una acción atómica en un proceso es elegible si es la próxima acción atómica en el proceso que será ejecutada. Si hay varios procesos ⇒ hay varias acciones atómicas elegibles. Una política de scheduling determina cuál será la próxima en ejecutarse.

Ejemplo: Si la política es asignar un procesador a un proceso hasta que termina o se demora. ¿Qué podría suceder en este caso?

```
bool continue = true;
co while (continue); // continue = false; oc
```

Definicion

Fairness Incondicional: Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada. En el ejemplo anterior, RR es incondicionalmente fair en monoprocesador, y la ejecución paralela lo es en un multiprocesador.

Definicion

Fairness Débil: Una política de scheduling es débilmente fair si :
1. Es incondicionalmente fair y
2. Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece true hasta que es vista por el proceso que ejecuta la acción atómica condicional.

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

Definicion

Fairness Fuerte: Una política de scheduling es fuertemente fair si:
1. Es incondicionalmente fair y
2. Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

No es simple tener una política que sea práctica y fuertemente fair. En el ejemplo anterior, con 1 procesador, una política que alterna las acciones de los procesos sería fuertemente fair, pero es impráctica. Round-robin es práctica pero no es fuertemente fair.

Clase 2

Sincronización por variables compartidas

Herramientas para la concurrencia

- Memoria Compartida
  - Variables compartidas
  - Semáforos
  - Monitores

- Memoria distribuida (pasaje de mensajes)
  - Mensajes asincrónicos
  - Mensajes sincrónicos
  - Remote Procedure Call (RPC)
  - Rendezvous

Locks y barreras

> **Definicion**
>
> Problema de la Sección Crítica: implementación de acciones atómicas en software (locks).

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

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

- Ventaja de implementarse con instrucciones de cualquier procesador.
- Ineficiente en multiprogramación (cuando varios procesos comparten el procesador y la ejecución es intercalada).
- Aceptable si cada proceso ejecuta en su procesador

El problema de la Sección crítica

```
process SC[i=1 to n]
{ while (true)
    { protocolo de entrada; ⇔ <
      sección crítica; ⇔ SC
      protocolo de salida; ⇔ >
      sección no crítica;
    }
}
```

Las soluciones a este problema pueden usarse para implementar sentencias **await** arbitrarias. ¿Qué propiedades deben satisfacer los protocolos de entrada y salida?.

Propiedades a cumplir

> **Important**

Exclusión mutua: A lo sumo un proceso está en su SC.
Ausencia de Deadlock (Livelock): si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.
Ausencia de Demora Innecesaria: si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.
Eventual Entrada: un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

• Solución trivial ⟨SC⟩. Pero, ¿cómo se implementan los ⟨⟩ ?

Implementación de sentencias await

- Cualquier solución al problema de la SC se puede usar para implementar una acción atómica incondicional ⟨S⟩ ; ⇒SCEnter ; S; SCExit
- Para una acción atómica condicional ⟨await (B) S;⟩ ⇒ SCEnter ; while (not B) {SCExit; SCEnter;} S; SCExit;
- • Si S es skip, y B cumple ASV, ⟨await (B);⟩ puede implementarse por medio de ⇒ while (not B) skip;

Correcto, pero ineficiente: un proceso está spinning continuamente saliendo y entrando a SC hasta que otro altere una variable referenciada en B.

- Para reducir contención de memoria ⇒ SCEnter ; while (not B) {SCExit; Delay; SCEnter;} S; SCExit;

Solución hardware: deshabilitar interrupciones

```
process SC[i=1 to n] {
    while (true) {
        deshabilitar interrupciones; # protocolo de entrada
        sección crítica;
        habilitar interrupciones; # protocolo de salida
        sección no crítica;
    }
}
```

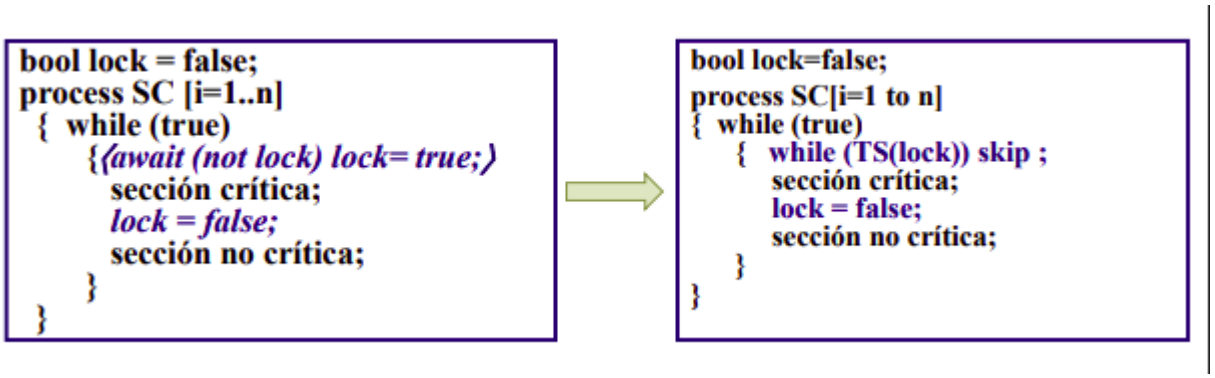
- Solución correcta para una máquina monoprocesador.
- Durante la SC no se usa la multiprogramación → penalización de performance
- La solución no es correcta en un multiprocesador.

Solución de "grano fino": Spin Locks

- Objetivo: hacer "atómico" el await de grano grueso.
- Idea: usar instrucciones como ***Test & Set*** (TS), ***Fetch & Add*** (FA) o ***Compare & Swap***, disponibles en la mayoría de los procesadores.

¿Como funciona ***Test & Set***?

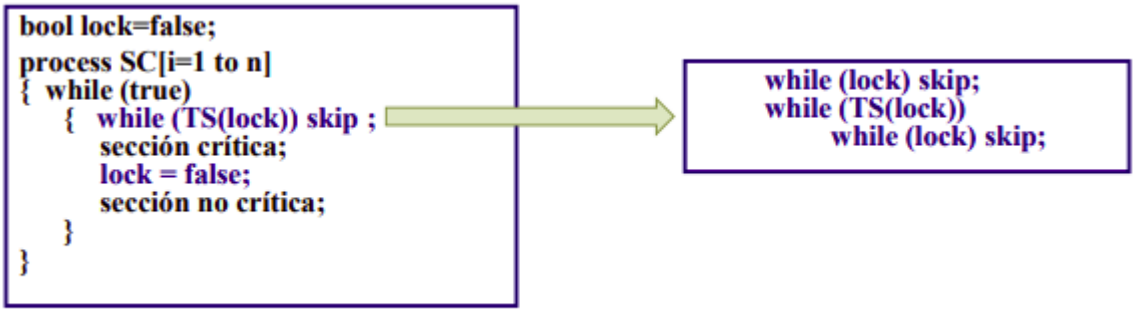
```
bool TS(bool ok);
{<bool inicial = ok;
 ok = true;
 return inicial;>
}
```



Solución tipo "spin locks": los procesos se quedan iterando (spinning) mientras esperan que se limpie lock.

Cumple las 4 propiedades si el scheduling es fuertemente fair.
Una política débilmente fair es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC).

TS escribe siempre en lock aunque el valor no cambie ⇒ Mejor Test-and-Test-and-Set



Memory contention se reduce, pero no desaparece. En particular, cuando lock pasa a false posiblemente todos intenten hacer TS.

Solución Fair: algoritmo Tie-Breaker

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

Algoritmo Tie-Breaker (2 procesos): protocolo de SC que requiere scheduling sólo débilmente fair y no usa instrucciones especiales ⇒ más complejo.
Usa una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la sección crítica, y una variable adicional para romper empates, indicando qué proceso fue el último en comenzar dicha entrada ⇒ esta última variable es

compartida y de acceso protegido.
Demora (quita prioridad) al último en comenzar su protocolo de entrada (entry protocol).

Solución de “Grano Grueso” al Algoritmo **Tie-Breaker**

```
bool in1 = false, in2 = false;
int ultimo = 1;
process SC1 {
    while (true) {  ultimo = 1; in1 = true;
                    ⟨await (not in2 or ultimo==2);⟩
                    sección crítica;
                    in1 = false;
                    sección no crítica;
                }
}
process SC2 {
    while (true) { ultimo = 2; in2 = true;
                    ⟨await (not in1 or ultimo==1);⟩
                   sección crítica;
                   in2 = false;
                   sección no crítica;
    }
}
```

Solución de “Grano Fino” al Algoritmo **Tie-Breaker**

```
bool in1 = false, in2 = false;
int ultimo = 1;
process SC1 {
    while (true) { in1 = true; ultimo = 1;
                   while (in2 and ultimo == 1) skip;
                   sección crítica;
                   in1 = false;
                   sección no crítica;
    }
}
process SC2 {
    while (true) { in2 = true; ultimo = 2;
                   while (in1 and ultimo == 2) skip;
                   sección crítica;
                   in2 = false;
                   sección no crítica;
    }
}
```

Generalización a n procesos:

- Si hay n procesos, el protocolo de entrada en cada uno es un loop que itera a través de n-1 etapas.

- En cada etapa se usan instancias de tie-breaker para dos procesos para determinar cuáles avanzan a la siguiente etapa.
- Si a lo sumo a un proceso a la vez se le permite ir por las n-1 etapas ⇒ a lo sumo uno a la vez puede estar en la SC.

```
int in[1:n] = ([n] 0), ultimo[1:n] = ([n] 0);
process SC[i = 1 to n] {
   while (true) {
      for [j = 1 to n] {     # protocolo de entrada
            # el proceso i está en la etapa j y es el último
            in[i] = j; ultimo[j] = i;
            for [k = 1 to n st i <> k] {
                # espera si el proceso k está en una etapa más alta
                # y el proceso i fue el último en entrar a la etapa j
                while (in[k] >= in[i] and ultimo[j]==i) skip;
             }
      }
      sección crítica;
      in[i] = 0;
      sección no crítica;
   }
}
```

Solución Fair: algoritmo Ticket

Tie-Breaker n-procesos ⇒ complejo y costoso de tiempo

> **Definicion**
>
> Algoritmo Ticket: se reparten números y se espera a que sea el turno. Los procesos toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los procesos con número más chico han sido atendidos.

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

{ TICKET: proximo > 0 ^ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]== proximo) ^
   (turno[i] >0) ⇒ (∀j: 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) )}

process SC [i: 1..n]
{ while (true)
   {  < turno[i] = numero;  numero = numero +1; >
      < await turno[i] == proximo; >
      sección crítica;
      < proximo = proximo + 1; >
      sección no crítica;
   }
}
```

Potencial problema: los valores de próximo y turno no son ilimitados. En la práctica, podrían resetearse a un valor chico (por ejemplo, 1).

Cumplimiento de las propiedades:

- El predicado ***TICKET*** es un invariante global, pues ***número*** es leído e incrementado en una acción atómica y ***próximo*** es incrementado en una acción atómica ⇒ hay a lo sumo un

proceso en la SC.
- La ausencia de deadlock y de demora innecesaria resultan de que los valores turno son únicos.
- Con scheduling débilmente fair se asegura eventual entrada.

El **await** puede implementarse con busy waiting (la expresión booleana referencia una sola variable compartida).
El incremento de ***proximo*** puede ser un load/store normal (a lo sumo un proceso puede estar ejecutando su protocolo de salida).

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );
process SC [i: 1..n]
{ while (true)
    {   <turno[i] = numero; numero = numero +1>
        while (turno[i] <> proximo) skip;
        sección crítica;
        proximo = proximo + 1;
        sección no crítica;
    }
}
```

¿Cómo se implementa la primera acción atómica donde se asigna el número? • Sea Fetch-and-Add una instrucción con el siguiente efecto:
FA(var,incr): < temp = var; var = var + incr; return(temp) >

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );
process SC [i: 1..n]
{ while (true)
    { turno[i] = FA (numero, 1);
      while (turno[i] <> proximo) skip;
      sección crítica;
      proximo = proximo + 1;
      sección no crítica;
    }
}
```

Solución Fair: algoritmo Bakery

Ticket ⇒ si no existe FA se debe simular con una SC y la solución puede no ser fair.

> **Definicion**
>
> Algoritmo Bakery: Cada proceso que trata de ingresar recorre los números de los demás y se auto asigna 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 Bakery es más complejo, pero es fair y no requiere instrucciones especiales.
- No requiere un contador global proximo que se “entrega” a cada proceso al llegar a la SC.

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

{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀j : 1 ≤ j ≤ n, j ≠ i:
turno[j] = 0 ∨ turno[i] < turno[j] ) ) }

process SC[i = 1 to n]
{  while (true)
       {    ⟨ turno[i] = max(turno[1:n] + 1; ⟩
            for [j = 1 to n st j <> i]  ⟨ await (turno[j] == 0 or turno[i] < turno[j]); ⟩
            sección crítica
            turno[i] = 0;
            sección no crítica
       }
}
```

Esta solución de grano grueso no es implementable directamente:

- La asignación a turno[i] exige calcular el máximo de n valores.
- El await referencia una variable compartida dos veces

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

{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀j : 1 ≤ j ≤ n, j ≠ i:
turno[j] = 0 ∨ turno[i] < turno[j] ) ) }

process SC[i = 1 to n]
{  while (true)
       {  turno[i] = 1;   //indica que comenzó el protocolo de entrada
          turno[i] = max(turno[1:n]) + 1;
          for [j = 1 to n st j != i]        //espera su turno
                while (turno[j] != 0)  and (  (turno[i],i) > (turno[j],j) ) →  skip;
           sección crítica
           turno[i] = 0;
           sección no crítica
       }
}
```

Sincronización Barrier

> **Definicion**
>
> 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).

Contador compartido

n procesos necesitan encontrarse en una barrera:

- Cada proceso incrementa una variable Cantidad al llegar.
- Cuando Cantidad es n los procesos pueden pasar

```
int cantidad = 0; process Worker[i=1 to n]
{ while (true)
    { código para implementar la tarea i;
      ⟨cantidad = cantidad + 1;⟩
      ⟨await (cantidad == n);⟩
     }
}
```

Flags y coordinadores

- Si no existe FA → Puede distribuirse Cantidad usando n variables (arreglo arribo[1..n]).
- El await pasaría a ser: ⟨await (arribo[1] + ... + arribo[n] == n);⟩
- Reintroduce contención de memoria y es ineficiente.

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

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);
process Worker[i=1 to n]
{ while (true)
    { código para implementar la tarea i;
      arribo[i] = 1;
      while (continuar[i] == 0) skip;
      continuar[i] = 0;
    }
}
process Coordinador
{ while (true)
    { for [i = 1 to n]
        { while (arribo[i] == 0) skip;
          arribo[i] = 0;
        }
      for [i = 1 to n] continuar[i] = 1;
    }
}
```

Árboles

Problemas:

- Requiere un proceso (y procesador) extra.
- El tiempo de ejecución del coordinador es proporcional a n.
  Posible solución:
- Combinar las acciones de Workers y Coordinador, haciendo que cada Worker sea también Coordinador.
- Por ejemplo, Workers en forma de árbol: las señales de arribo van hacia arriba en el árbol, y las de continuar hacia abajo ⇒ combining tree barrier (más eficiente para n grande).

Barrera Simétrica

- En combining tree barrier los procesos juegan diferentes roles.
- Una Barrera Simétrica para n procesos se construye a partir de pares de barreras simples para dos procesos:

```
W[i]:: ⟨ await (arribo[i] == 0); ⟩        W[j]:: ⟨ await (arribo[j] == 0); ⟩
       arribo[i] = 1;                            arribo[j] = 1;
       ⟨ await (arribo[j] == 1); ⟩               ⟨ await (arribo[i] == 1); ⟩
       arribo[j] = 0;                            arribo[i] = 0;
```

- ¿Cómo se combinan para construir una barrera n proceso? Worker[1:n] arreglo de procesos. Si n es potencia de 2 ⇒Butterfly Barrier.
- log2 n etapas: cada worker sincroniza con uno distinto en cada etapa.
- En la etapa s, un worker sincroniza con otro a distancia 2 s-1 .
- Cuando cada worker pasó log2 n etapas, todos pueden seguir.

Butterfly barrier

```
int E = log(N);
int arribo[1:N] = ([N] 0);

process P[i=1..N]
{  int j;
   while (true)
     {  //Sección de código anterior a la barrera.
        //Inicio de la barrera
        for (etapa = 1; etapa <= E; etapa++)
          {  j =  (i-1) XOR (1<<(etapa-1));    //calcula el proceso con cual sincronizar
             while (arribo[i] == 1) → skip;
             arribo[i] = 1;
             while (arribo[j] == 0) → skip;
             arribo[j] = 0;
          }
        //Fin de la barrera
        //Sección de código posterior a la barrera.
     }
}
```

Defectos de la sincronización por busy waiting

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

Necesidad de herramientas para diseñar protocolos de sincronización.

Clase 3

Defectos de la sincronización por Busy Waiting

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

⇒ Necesidad de herramientas para diseñar protocolos de sincronización

Semáforos

> **Definición**
>
> Semáforo ⇒ instancia de un tipo de datos abstracto (o un objeto) con sólo operaciones (métodos) atómicas: P y V.
> Internamente el valor de un semáforo es un centro no negativo:
>
> - V -> Señala la **ocurrencia de un evento** (incrementa).
> - P -> Se usa para **demorar** un proceso **hasta que ocurra un evento** (decrementa)

- Analogía con la sincronización del tránsito para evitar colisiones.
- Permiten proteger Secciones Críticas y pueden usarse para implementar Sincronización por Condición.

Operaciones básicas

Declaraciones:

> **Important**
>
> sem s; -> No. Si o si deben inicializar en declaración

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

Semáforo general (o counting semaphore)

- P(s): ⟨await (s > 0) s = s - 1;⟩
- V(s): ⟨ s = s + 1;⟩

**Semáforo binario

- P(b): ⟨await(b > 0) b = b - 1;⟩
- V(b): ⟨await(b > 0) b = b + 1;⟩

Si la implementación de la demora por operaciones P se produce sobre una cola, las operaciones son fair.
(EN LA MATERIA NO SE PUEDE SUPONER ESTE TIPO DE IMPLEMENTACIÓN)

Problemas básicos y técnicas

Sección crítica: Exclusión Mutua

```
bool lock=false;

process SC[i=1 to n]
{  while (true)
     {  ⟨await (not lock) lock = true;⟩
        sección crítica;
        lock = false;
        sección no crítica;
   }
}
```

Cambio de variable

```
bool free = true;

process SC[i=1 to n]
{  while (true)
     {  ⟨await (free) free = false;⟩
        sección crítica;
        free = true;
        sección no crítica;
   }
}
```

Podemos representar free con un entero, usar 1 para true y 0 para false ⇒ se puede asociar a las operaciones soportadas por los semáforos.

```
int free = 1; process SC[i=1 to n]
{ while (true)
    {  ⟨await (free==1) free = 0;⟩
       sección crítica;
       free = 1;
       sección no crítica;
    }
}
```

```
int free = 1;

process SC[i=1 to n]
{  while (true)
     {  <await (free==1) free = 0;>
        sección crítica;
        free = 1;
        sección no crítica;
   }
}
```

```
int free = 1;

process SC[i=1 to n]
{  while (true)
     {  <await (free > 0) free = free - 1;>
        sección crítica;
        <free = free + 1>;
        sección no crítica;
   }
}
```

```
sem free= 1;
process SC[i=1 to n]
{  while (true)
     {  P(free);
        sección crítica;
        V(free);
        sección no crítica;
   }
}
```

Es más simple que las soluciones ***busy waiting***.

¿Y si inicializo free= 0?

Barreras: señalización de eventos

- **Idea:** un semáforo para cada flag de sincronización. Un proceso setea el flag ejecutando **V**, y espera a que un flag sea seteado y luego lo limpia ejecutando **P**.
- **Barrera para dos procesos**: necesitamos saber cada vez que un proceso llego o parte de la barrera ⇒ relacionar los estados de los procesos

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

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

```
sem llega1=0, llega2=0;
process Worker1 {
    ..........
    V(llega1); P(llega2);
    ........
}
process Worker2 {
    ..........
    V(llega2); P(llega1);
    ........
}
```

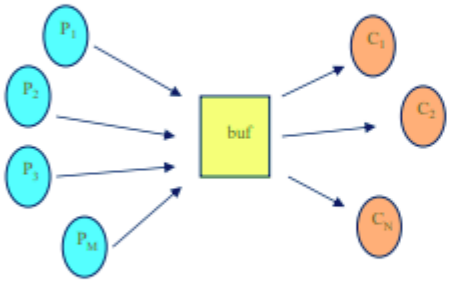
Productores y Consumidores: semáforos binarios divididos

> **Definicion**
>
> Semáforo Binario Dividido (Split Binary Semaphore). Los semáforos binarios b1 , ..., bn forman un SBS en un programa si el siguiente es un invariante global:
> SPLIT: 0 >= b1 , + ... + bn <= 1

- Los bi pueden verse como un único semáforo binario **b** que fue dividido en n semáforos binarios.
- Importantes por la forma en que pueden usarse para implementar EM (en general la ejecución de los procesos inicia con un **P** sobre un semáforo y termina con un **V** sobre otro de ellos).
- Las sentencias entre el **P** y el **V** ejecutan con exclusión mutua.

Ejemplo

Ejemplo: buffer unitario compartido con múltiples productores y consumidores. Dos operaciones: ***depositar*** y ***retirar*** que deben alternarse.




```
typeT buf;  sem vacio = 1, lleno = 0;

process Productor [i = 1 to M]
{ while(true)
    { …
      producir mensaje datos
      P(vacio); buf = datos; V(lleno); #depositar
    }
}

process Consumidor[j = 1 to N]
{ while(true)
    { P(lleno); resultado = buf;  V(vacio); #retirar
      consumir mensaje resultado
      …
    }
}
```

vacio y ***lleno*** (juntos) forman un ***"semáforo binario dividido"***.

Buffers Limitados: Contadores de Recursos

> **Definición**
>
> Contadores de Recursos: cada semáforo cuenta el número de unidades libres de un recurso determinado. Esta forma de utilización es adecuada cuando los procesos compiten por recursos de múltiples unidades.

Ejemplo: un buffer es una cola de mensajes depositados y aún no buscados. Existe UN productor y UN consumidor que depositan y retiran elementos del buffer.

```
typeT buf[n];  int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;
process Productor
{ while(true)
    { …
      producir mensaje datos
      P(vacio); buf[libre] = datos; libre = (libre+1) mod n;  V(lleno); #depositar
    }
}
process Consumidor
{ while(true)
    { P(lleno); resultado = buf[ocupado];  ocupado = (ocupado+1) mod n; V(vacio); #retirar
      consumir mensaje resultado
      …
    }
}
```

- vacio cuenta los lugares libres, y lleno los ocupados.
- depositar y retirar se pudieron asumir atómicas pues sólo hay un productor y un consumidor.

- ¿Qué ocurre si hay más de un productor y/o consumidor?

Si hay más de un productor y/o más de un consumidor, las operaciones de depositar y retirar en sí mismas son SC y deben ejecutar con Exclusión Mutua ¿Cuáles serían las consecuencias de no protegerlas?
Si no se protege cada slot, podría retirarse dos veces el mismo dato o perderse datos al sobrescribirlo..

Varios procesos compitiendo por recursos compartidos

- Problema de varios procesos (P) y varios recursos (R) cada uno protegido por un lock.
- Un proceso debe adquirir los locks de todos los recursos que necesita.
- Puede caerse en deadlock cuando varios procesos compiten por conjuntos superpuestos de recursos.
- Por ejemplo: cada P[i] necesita R[i] y R[(i+1) mod n]

Técnica Passing the Baton

> **Definicion**
>
> Passing the baton: técnica general para implementar sentencias await.
> 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 (sale de la SC), pasa el baton (control) a otro proceso. Si ningún proceso está esperando por el baton (es decir esperando entrar a la SC) el baton se libera para que lo tome el próximo proceso que trata de entrar.

- En algunos casos, await puede ser implementada directamente usando semáforos u otras operaciones primitivas. Pero no siempre…
- En el caso de las guardas de los await en la solución anterior, se superponen en que el protocolo de entrada para escritores necesita que tanto nw como nr sean 0, mientras para lectores sólo que nw sea 0.
- Ningún semáforo podría discriminar entre estas condiciones → Passing the baton

La sincronización se expresa con sentencias atómicas de la forma:
F1: $\langle Si \rangle$ o $F2 : \langle \text{await } (Bj) Sj \rangle$
Puede hacerse con semáforos binarios divididos (SBS).
e semáforo binario inicialmente 1 (controla la entrada a sentencias atómicas). Utilizamos un semáforo bj y un contador dj cada uno con guarda diferente Bj ; todos inicialmente 0.
bj se usa para demorar procesos esperando que Bj sea true.
dj es un contador del número de procesos demorados sobre bj .
e y los bj se usan para formar un SBS: a lo sumo uno a la vez es 1, y cada camino de ejecución empieza con un P y termina con un único V.

F_1 : P(e);
S_i;
SIGNAL;

⟨ S_i ⟩

F_2 : P(e);
if (not B_j) {$d_j = d_j + 1$; V(e); P(b_j); }
S_j;
SIGNAL

⟨ await (B_j) S_j ⟩

SIGNAL: if (B_1 and $d_1 > 0$) {$d_1 = d_1 - 1$; V(b_1)}
□ ...
□ (B_n and $d_n > 0$) {$d_n = d_n - 1$; V(b_n)}
□ else V(e);
fi

Alocación de Recursos y Scheduling

- Problema: decidir cuándo se le puede dar a un proceso determinado acceso a un recurso.
- Recurso: cualquier objeto, elemento, componente, dato, SC, por la que un proceso puede ser demorado esperando adquirirlo.
- Definición del problema: procesos que compiten por el uso de unidades de un recurso compartido (cada unidad está libre o en uso).
  - request (parámetros): ⟨await (request puede ser satisfecho) tomar unidades;⟩
  - release (parámetros): ⟨retornar unidades;⟩
- Puede usarse Passing the Baton:
  - request (parámetros):
    P(e);
    if (request no puede ser satisfecho)
    DELAY;
    tomar las unidades;
    SIGNAL;
  - release (parámetros):
    P(e);
    retornar unidades;
    SIGNAL;

Alocación Shortest-Job-Next(SJN)

- Varios procesos que compiten por el uso de un recurso compartido de una sola unidad.
- request (tiempo,id). Si el recurso está libre, es alocado inmediatamente al proceso id; sino, el proceso id se demora.
- release ( ). Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) con el mínimo valor de tiempo. Si dos o más procesos tienen el mismo valor de tiempo, el recurso es alocado al que esperó más.
- SJN minimiza el tiempo promedio de ejecución, aunque es unfair (¿por qué?). Puede mejorarse con la técnica de aging (dando preferencia a un proceso que esperó mucho

tiempo).

- Para el caso general de alocación de recursos (NO SJN):
  bool libre = true;
  request (tiempo,id): ⟨await (libre) libre = false;⟩
  release (): ⟨libre = true;⟩
- En SJN, un proceso que invoca a request debe demorarse hasta que el recurso esté libre y su pedido sea el próximo en ser atendido de acuerdo a la política. El parámetro tiempo entra en juego sólo si un pedido debe ser demorado.
- En DELAY un proceso:
  - Inserta sus parámetros en un conjunto, cola o lista de espera (pares).
  - Libera la SC ejecutando V(e).
  - Se demora en un semáforo hasta que request puede ser satisfecho.
- En SIGNAL un proceso:
  - Cuando el recurso es liberado, si pares no está vacío, el recurso es asignado a un proceso de acuerdo a SJN.
- Cada proceso tiene una condición de demora distinta, dependiendo de su posición en pares. El proceso id se demora sobre el semáforo b[id]

```
bool libre = true;  Pares = set of (int, int) = ∅;  sem e = 1, b[n] = ([n] 0);

request(tiempo,id):   P(e);
                      if (! libre){ insertar (tiempo, id) en Pares; V(e); P(b[id]); }
                      libre = false;
                      V(e);

release( ):           P(e);
                      libre = true;
                      if (Pares ≠ ∅ ) { remover el primer par (tiempo,id) de Pares; V(b[id]); }
                      else V(e);
```

s es un semáforo privado si exactamente un proceso ejecuta operaciones P sobre s. Resultan útiles para señalar procesos individuales. Los semáforos b[id] son de este tipo.

Clase 4

Monitores

Conceptos básicos

Definicion

Monitores: módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos
Mecanismos de abstracción de datos:

- Encapsulan las representaciones de recursos
- Brindan un conjunto de operaciones que son los únicos medios para manipular los recursos

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

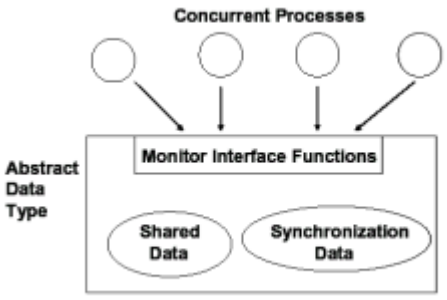
> **Nota**
>
> Exclusión mutua ⇒ implícita asegurando que los procedures en el mismo monitor no ejecutan concurrentemente

> **Nota**
>
> Sincronización por condición ⇒ explícita con variables condición

Programa concurrente ⇒ procesos activos y monitores pasivos. Dos procesos interactúan invocando procedures de un monitor.

Ventajas:

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



Notación

- Un monitor agrupa la representación y la implementación de un recurso compartido, se distingue a un monitor de un TAD en procesos secuenciales en que es compartido por procesos que se ejecutan concurrentemente. Tiene interfaz y cuerpo:
  - La interfaz especifica operaciones que brinda el recurso.
  - El cuerpo tiene variables que representan el estado del recurso y procedures que implementan las operaciones de la interfaz.
- Sólo los nombres de los procedures son visibles desde afuera. Sintácticamente, los llamados al monitor tienen la forma:
  - **NombreMonitor.op*i*(argumentos)**
- Los procedures pueden acceder sólo a las variables permanentes, sus variables locales, y parámetros que le sean pasados en la invocación
- El programador de un monitor no puede conocer a priori el orden del llamado

```
monitor NombreMonitor {
    declaraciones de variables permanentes;
    código de inicialización
    procedure op1 (par. formales1 )
    { cuerpo de op1
    }
    .......
    procedure opn (par. formalesn )
    { cuerpo de opn
    }
}
```

Sincronización

La sincronización por condición es programada con variables condición --> cond cv;
El valor asociado a cv es una cola de procesos demorados, no visible directamente al programador. Operaciones sobre las variables condición:

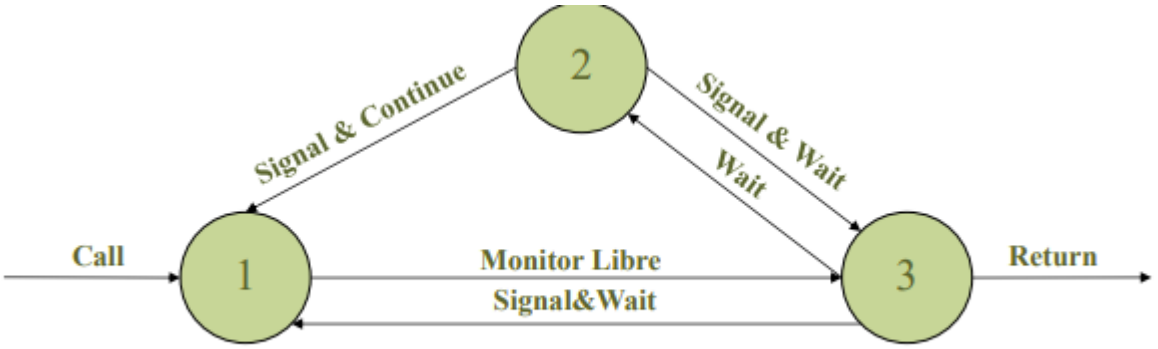
- wait(cv) → el proceso se demora al final de la cola de cv y deja el acceso exclusivo al monitor.
- signal(cv) → despierta al proceso que está al frente de la cola (si hay alguno) y lo saca de ella. El proceso despertado recién podrá ejecutar cuando requiera el acceso exclusivo al monitor.
- signal_all(cv) → despierta todos los procesos demorados en cv, quedando vacía la cola asociada a cv.
- Disciplinas de señalización:
  - Signal and continued ⇒ es el utilizado en la materia.
  - Signal and wait.

Operaciones adicionales

Operaciones adicionales que NO SON USADAS EN LA PRÁCTICA sobre las variables condición:

- empty(cv) → retorna true si la cola controlada por cv está vacía.
- wait(cv, rank) → el proceso se demora en la cola de cv en orden ascendente de acuerdo al parámetro rank y deja el acceso exclusivo al monitor.
- minrank(cv) → función que retorna el mínimo ranking de demora.

Signal and continue vs Signal and Wait



1. Espera acceso al monitor.
2. Cola por Variable Condición.
3. Ejecutando en el Monitor.

Resumen: diferencia entre las disciplinas de señalización

- Signal and Continued: 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.

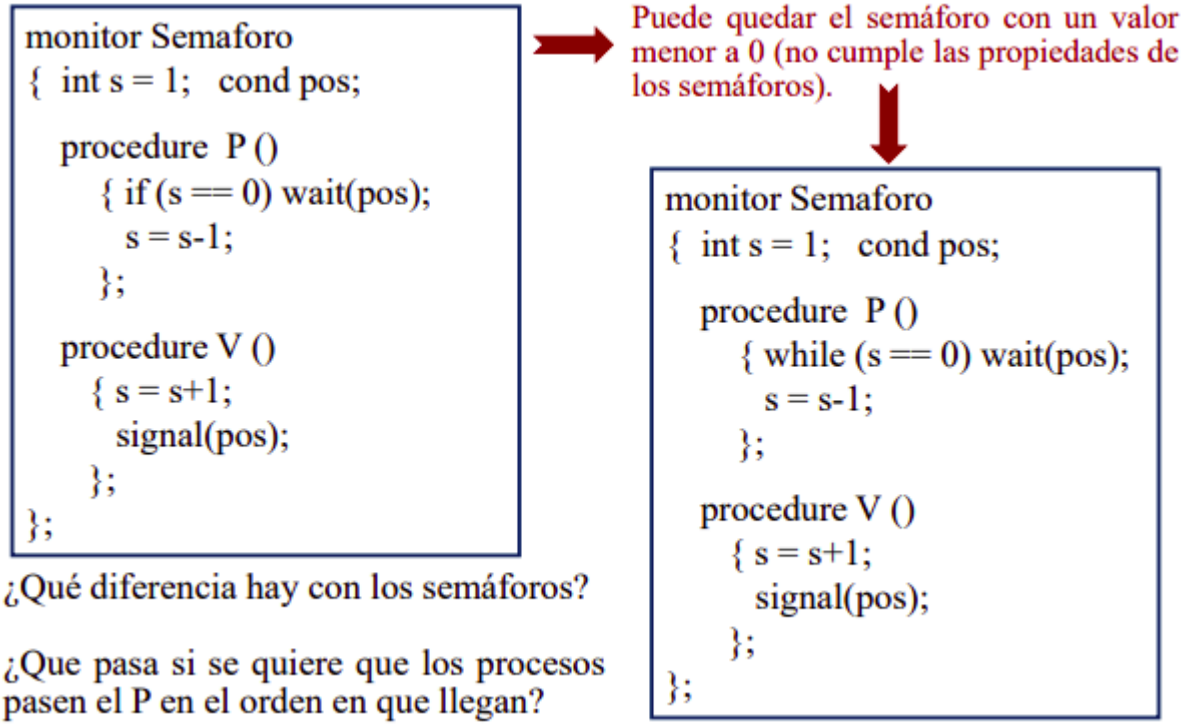
Resumen: diferencia entre wait/signal con P/V

Wait	P
El proceso siempre se duerme	El proceso solo se duerme si el semáforo es 0

Signal	V
Si hay procesos dormidos despierta al primero de ellos. En caso contrario no tiene efecto posterior	Incrementa el semáforo para que un proceso dormido o que hará un P continúe. No sigue ninún orden al despertarlos

Ejemplos y técnicas

Simulación de semáforos: condición básica



Tecnicas de sincronización

Simulación de semáforos: passing the conditions

```
monitor Semaforo
{ int s = 1; cond pos;
  procedure P ()
  { if (s == 0) wait(pos)
    else s = s-1;
   };
  procedure V () {
      if (empty(pos) ) s = s+1
      else signal(pos);
  };
};
```

Como resolver este problema al no contar con la sentencia empty.

```
monitor Semaforo {
    int s = 1,
    espera = 0;
    cond pos;
    procedure P () {
        if (s == 0) { espera ++; wait(pos);}
        else s = s-1;
    };
    procedure V () {
        if (espera == 0 ) s = s+1
        else { espera --; signal(pos);}
```

```
    };
};
```

Alocación SJN: Wait con Prioridad

```
monitor Shortest_Job_Next
 { bool libre = true;
   cond turno;

   procedure request (int tiempo) {
       if (libre) libre = false;
       else wait (turno, tiempo);
   };

   procedure release () {
       if (empty(turno)) libre = true
       else signal(turno);
   };
}
```

- Se usa wait con prioridad para ordenar los procesos demorados por la cantidad de tiempo que usarán el recurso.
- Se usa empty para determinar si hay procesos demorados.
- Cuando el recurso es liberado, si hay procesos demorados se despierta al que tiene mínimo rank.
- Wait no se pone en un loop pues la decisión de cuándo puede continuar un proceso la hace el proceso que libera el recurso.

¿Como resolverlo sin wait con prioridad?

Alocación SJN: Variables Condición Privadas

- Se realiza Passing the Condition, manejando el orden explícitamente por medio de una cola ordenada y variables condición privadas

```
monitor Shortest_Job_Next {
    bool libre = true;
    cond turno[N];
    cola espera;
    procedure request (int id, int tiempo) {
        if (libre) libre = false
        else {
            insertar_ordenado(espera, id, tiempo);
            wait (turno[id]);
        };
    };
    procedure release () {
        if (empty(espera)) libre = true
```

```
        else {
            sacar(espera, id);
            signal(turno[id]);
        };
    };
}
```

Passing the Condition

```
monitor Controlador_RW
{ int nr = 0, nw = 0, dr = 0, dw = 0;
cond ok_leer, ok_escribir


    procedure pedido_leer( ){
     if (nw > 0){
       dr = dr +1;
       wait (ok_leer);
      }
      else nr = nr + 1;
     }
    procedure libera_leer( ){
      nr = nr - 1;
      if (nr == 0 and dw > 0)
         { dw = dw - 1;
           signal(ok_escribir)
           nw = nw + 1;
         }
    }
    procedure pedido_escribir( )
         { if (nr>0 OR nw>0)
        { dw = dw +1;
         wait (ok_escribir);
         }
         else nw = nw + 1;
         }
    procedure libera_escribir( )
    { if (dw > 0){
         dw = dw -1;
         signal (ok_escribir);
      }
      else {
         nw = nw -1;
         if (dr > 0)
            { nr = dr;
             dr = 0;
             signal_all (ok_leer);
             }
       }
```

```
    }
}
```

Diseño de un reloj lógico

Covering conditions

```
monitor Diseño de un reloj lógico Timer
{ int hora_actual = 0;
  cond chequear;
  procedure demorar(int intervalo){
      int hora_de_despertar;
      hora_de_despertar=hora_actual+intervalo;
      while (hora_de_despertar>hora_actual)
          wait(chequear);
    }
 procedure tick( ){
     hora_actual = hora_actual + 1;
     signal_all(chequear);
 }
}
```

- Timer que permite a los procesos dormirse una cantidad de unidades de tiempo.
- Ejemplo de controlador de recurso (reloj lógico) con dos operaciones:
  - demorar(intervalo): demora al llamador durante intervalo ticks de reloj.
  - tick: incrementa el valor del reloj lógico. Es llamada por un proceso que es despertado periódicamente por un timer de hardware y tiene alta prioridad de ejecución.

Ineficiente → mejor usar wait con prioridad o variables condition privadas

Wait con prioridad

El mismo ejemplo anterior del reloj lógico utilizando wait con prioridad:

```
monitor Timer
{ int hora_actual = 0;
 cond espera;
 procedure demorar(int intervalo)
 { int hora_de_despertar;
 hora_de_despertar = hora_actual + intervalo;
wait(espera, hora_a_despertar);
}
 procedure tick( )
 { hora_actual = hora_actual + 1;
 while (minrank(espera) <= hora_actual)
 signal (espera);
```

```
 }
}
```

Variables conditions privadas

El mismo ejemplo anterior del reloj lógico utilizando variables conditions privadas:

```
monitor Timer
{ int hora_actual = 0;
 cond espera[N];
 colaOrdenada dormidos;
 procedure demorar(int intervalo, int id)
 { int hora_de_despertar;
 hora_de_despertar = hora_actual + intervalo;
Insertar(dormidos, id, hora_de_despertar);
wait(espera[id]);
}
 procedure tick( )
 { int aux, idAux;
 hora_actual = hora_actual + 1;
 aux = verPrimero (dormidos);
 while (aux <= hora_actual)
 { sacar (dormidos, idAux)
 signal (espera[idAux]);
 aux = verPrimero (dormidos);
 }
 }
}
```

Clase 5

Programación concurrente en memoria distribuida

Conceptos generales

- Arquitecturas de memoria distribuida ⇒ *procesadores + memo local + red de comunicaciones + **mecanismo de comunicación / sincronizacion*** ⇒ **intercambio de mensajes**
- **Programa distribuido**: programa concurrente comunicado por mensajes. Supone la ejecución sobre una arquitectura de memoria distribuida, aunque puedan ejecutarse sobre una memoria compartida (o híbrida).
- **Primitivas de pasaje de mensajes**: interfaz con el sistema de comunicaciones ⇒ semáforos + datos + sincronización.
- Los procesos SOLO **comparten canales**(físicos o lógicos). Variantes para canales:
  - Mailbox, input port, link
  - Uni o bidireccionales
  - Sincrónicos o asincrónicos

Características

- Los canales son lo único que comparten los procesos
  - Variables locales a un proceso ("cuidador").
  - La exclusión mutua no requiere mecanismo especial.
  - Los procesos interactúan comunicándose.
  - Accedidos por primitivas de envío y recepción.
- Mecanismos para el Procesamiento Distribuido:
  - Pasaje de Mensajes Asincrónicos (PMA)
  - Pasaje de Mensajes SIncrónico (PMS)
  - Llamado a Procedimientos Remotos (RPC)
  - Rendezvous
- La sincronización de la comunicación interproceso depende del patrón de interacción:
  - Productores y consumidores (FIltros o pipes)
  - Clientes y servidores
  - Pares que interactúan

> Cada mecanismo es mas adecuado para determinados patrones

Relación entre mecanismos de sincronización

- Semáforos ⇒ mejora respecto de **busy waiting**
- Monitores ⇒ combinan Exclusión Mutua implícita y señalización explícita
- PM ⇒ extiende semáforos con datos
- RPC y rendezvous ⇒ combina la interface procedural de monitores con PM implícito

Pasaje de Mensajes Asincrónicos (PMA)

Canales

Uso en PMA

- **PMA** ⇒ canales = colas de mensajes enviados y aún no recibidos
- Declaración de canales ⇒ chan ch (id1: tipo1,...,idn:tipon)
  - **chan** entrada (char)
  - **chan** acceso_disco(INT cilindro, INT bloque, INT cant, CHAR* buffer)
  - **chan** resultado[n] (INT)
- Operación send ⇒ un proceso agrega un mensaje al final de la cola ("ilimitada") de un canal ejecutando un *send*, que no bloquea al emisor:
  - **send ch**(**expr1,...,exprn**)
- Operación recieve ⇒ un proceso recibe un mensaje desde un canal con recieve, que demora ("bloquea") al receptor hasta que un canal haya al menos un mensaje; luego toma el primero y lo almacena en variables locales:
  - **recieve ch**(**var1,...,varn**)
    Las variables del recieve deben tener los mismos tipos que la declaración del canal.

Recieve es una primitiva **bloqueante**, ya que produce un delay. Semántica: el proceso NO hace nada hasta recibir un mensaje en la cola correspondiente al canal.
NO es necesario hacer pooling

Características

Los canales son declarados globales a los procesos, ya que pueden ser compartidos. Según la forma en que se usan podría ser:

- Cualquier proceso puede enviar o recibir por alguno de los canales declarados. En este caso suelen denominarse mailboxes.
- En algunos casos un canal tiene un solo receptor y muchos emisores (input port).
- Si el canal tiene un único emisor y un único receptor se lo denomina link: provee un "camino" entre el emisor y sus receptores

Ejemplo

```
chan entrada(char), salida(char [CantMax]);
Process Carac_a_Linea
{ char linea [CantMax], int i = 0;
  WHILE (true)
    { receive entrada (linea[i]);
      WHILE (linea[i] □ CR and i < CantMax)
        { i := i + 1;
        receive entrada (linea[i]);
        }
    linea [i] := EOL;
    send salida(linea);
    i := 0;
    }
}
Process Proceso_1
{ char a;
  WHILE (true)
    { leer_carácter_por_teclado(a);
      send entrada(a);
    }
}
Process Proceso_2
{ char res[CantMax];
  WHILE (true)
    { receive salida(res);
    imprimir_en_pantalla(res);
    }
}
```

Clientes y servidores

Monitores activos

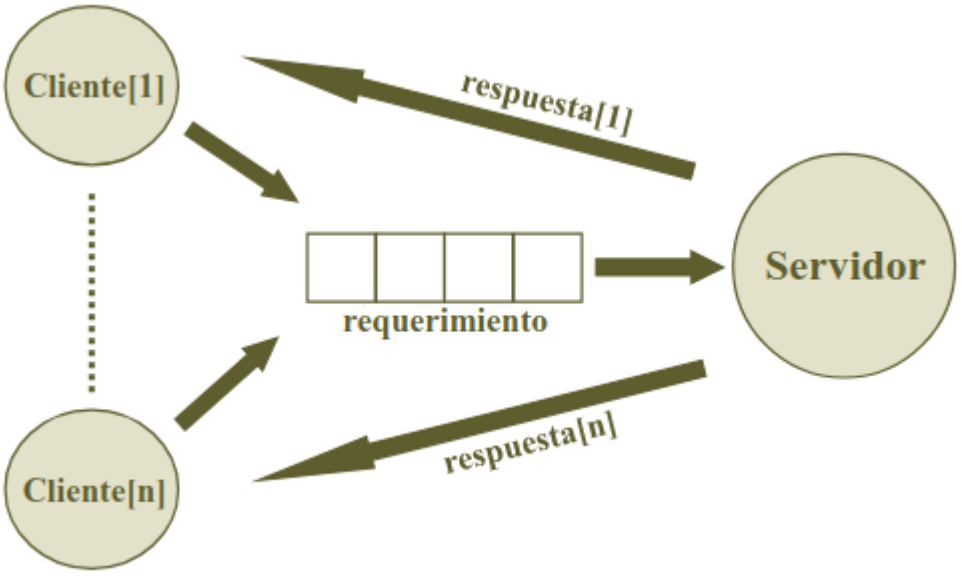
- Servidor: proceso que maneja pedidos (“requests”) de otros procesos clientes. ¿Cómo implementamos C/S con PMA?
- Dualidad entre monitores y PM: cada uno de ellos puede simular al otro.

> Monitor ⇒ manejador de recurso. Encapsula variables permanentes que registran el estado, y provee un conjunto de procedures. Los simulamos, usando procesos servidores y PM, como procesos activos en lugar de como conjuntos pasivos de procedures.

- Un Servidor es un proceso que maneja pedidos (requerimientos) de otros procesos clientes. Veremos cómo implementar Cliente/Servidor con PMA.
- Un proceso Cliente que envía un mensaje a un canal de requerimientos general, luego recibe el resultado desde un canal de respuesta propio (¿por qué?).
- En un sistema distribuido, lo natural es que el proceso Servidor resida en un procesador físico y M procesos Cliente residan en otros N procesadores (N<= M).

1 operación

- Para simular Mname, usamos un proceso server Servidor.
- Las variables permanentes serán variables locales de Servidor.
- Llamado: un proceso cliente envía un mensaje a un canal de requerimiento.
- Luego recibe el resultado por un canal de respuesta propio



Múltiples operaciones

- Podemos generalizar esta solución de C/S con una única operación para considerar múltiples operaciones.
- El IF del Servidor será un CASE con las distintas clases de operaciones.
- El cuerpo de cada operación toma datos de un canal de entrada en args y los devuelve al cliente adecuado en resultados

Múltiples operaciones y variables condición

- Hasta ahora el monitor no requería variables condición ya que el Servidor no requería demorar la atención de un pedido de servicio. Caso general: monitor con múltiples operaciones y con sincronización por condición. Para los clientes, la situación es transparente ⇒ cambia el servidor.
- Consideramos un caso específico de manejo de múltiples unidades de un recurso (ejemplos: bloques de memoria, impresoras).
  - Los clientes “adquieren” y devuelven unidades del recurso.
  - Las unidades libres se insertan en un “conjunto” sobre el que se harán las operaciones de INSERTAR y REMOVER.
  - El número de unidades disponibles es lo que “controla” nuestra variable de sincronización por condición.
- Caso en que el servidor tiene dos operaciones:
  - Si no hay unidades disponibles, el servidor no puede esperar hasta responder al pedido ⇒ debe salvarlo y diferir la respuesta.
  - Cuando una unidad es liberada, atiende un pedido salvado (si hay) enviando la unidad.
- El monitor y el Servidor muestran la dualidad entre monitores y PM: hay una correspondencia directa entre los mecanismos de ambos.
- La eficiencia de monitores o PM depende de la arquitectura física de soporte:
  - Con MC conviene la invocación a procedimientos y la operación sobre variables condición.
  - Con arquitecturas físicamente distribuidas tienden a ser más eficientes los mecanismos de PM.
- Dualidad entre Monitores y Pasaje de Mensajes

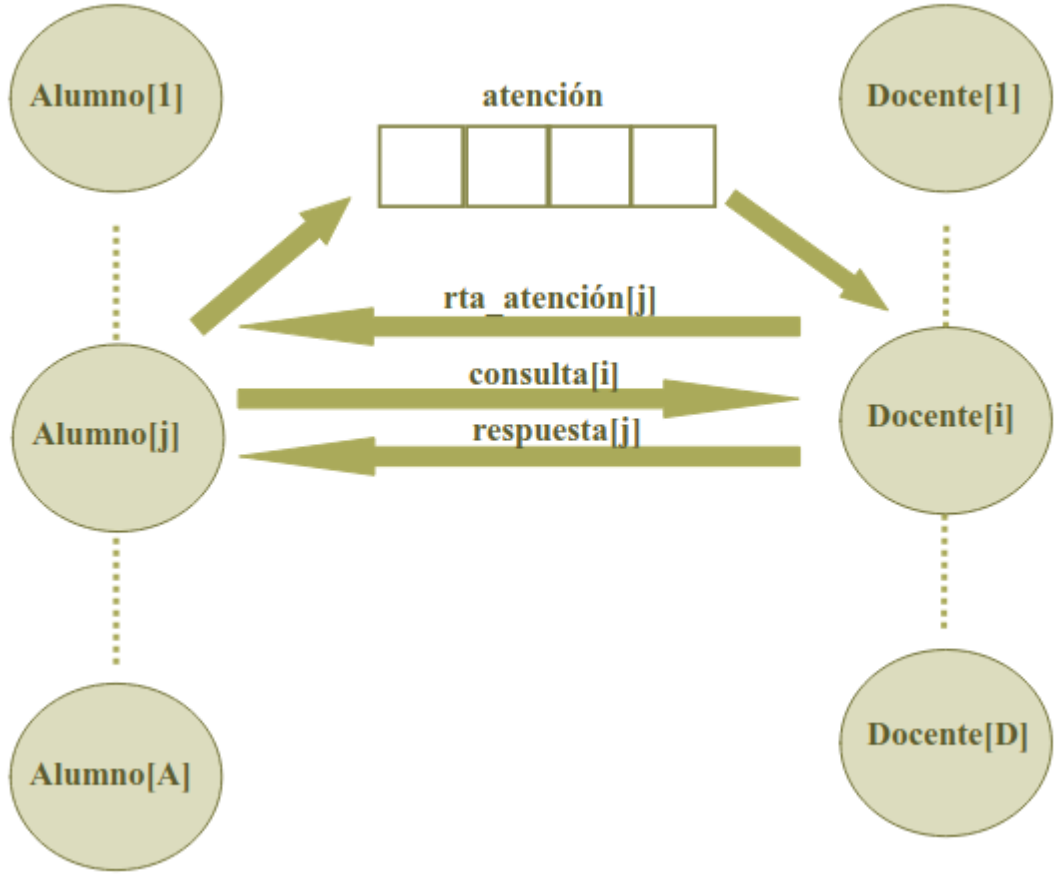
Programas con monitores	**Programas basados en PM**
Variables permanentes	Variables locales del servidor
Identificadores de procedures	Canal request y tipos de operación
Llamado a procedure	send request(): recieve respuesta
Entry del monitor	recieve request()
Retorno del procedure	send respuesta()
Sentencia wait	Salvar pedido pendiente
Sentencia signal	Recuperar/ procesar pedido pendiente
Cuerpos del procedure	Sentencias del "case" de acuerdo a la clase de operación

Continuidad Conversacional

- Existen A alumnos que hacen consultas a D docentes.
- El alumno espera hasta que un docente lo atiende, y a partir de ahí le comienza a realizar las diferentes consultas hasta que no le queden dudas.

- Los alumnos son los procesos “clientes”, y los docentes los procesos “Servidores”. Los procesos servidores son idénticos, y cualquiera de ellos que esté libre puede atender un requerimiento de un alumno.

> Todos los alumnos pueden pedir atención por un canal global y recibirán respuesta de un docente dado por un canal propio. ¿Por qué?



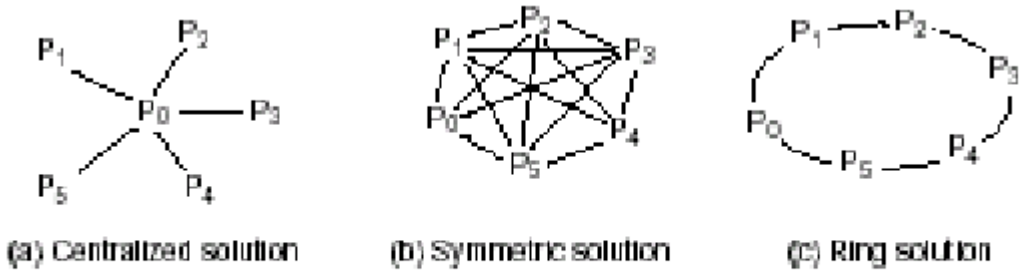
- Este ejemplo de interacción entre clientes y servidores se denomina continuidad conversacional (desde la solicitud de atención hasta la última consulta).
- atención es un canal compartido por el que cualquier Docente puede recibir. Si cada canal puede tener un solo receptor, se necesita otro proceso intermedio.

Pares (peers) interactuantes

Intercambio de valores

> Problema: cada proceso tiene un dato local V y los N procesos deben saber cuál es el menor y cuál el mayor de los valores.

Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: centralizado, simétrico y en anillo circular.



Solución centralizada

- Cada proceso tiene un valor V local. Al final todos los procesos deben conocer el mínimo y máximo valor de todo el sistema.
- La arquitectura centralizada es apta para una solución en que todos envían su dato local V al procesador central, éste ordena los N datos y reenvía la información del mayor y menor a todos los procesos ⇒ 2(N-1) mensajes.

Solución simétrica

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

Solución anillo circular

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

Comentarios sobre las soluciones

- Simétrica es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast). Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup.

- Centralizada y anillo usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance:
  - En centralizada, los mensajes al coordinador se envían casi al mismo tiempo sólo el primer receive del coordinador demora mucho.
  - En anillo, todos los procesos son productores y consumidores. El último tiene que esperar a que todos los otros (uno por vez) reciban un mensaje, hacer poco cómputo, y enviar su resultado.
    Los mensajes circulan 2 veces completas por el anillo ⇒ Solución inherentemente lineal y lenta para este problema.

Clase 6

Pasaje de Mensajes Sincrónicos (PMS)

Diferencia con PMA

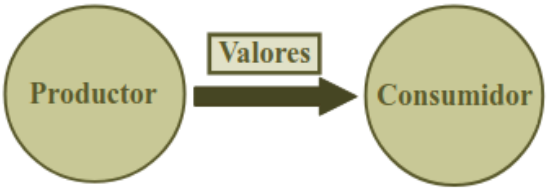
- Los canales son de tipo link o punto a punto (1 emisor y 1 receptor).
- La diferencia entre PMA y PMS es la primitiva de trasmisión Send. En PMS es bloqueante y la llamaremos (por ahora) sync_send.
  - El trasmisor queda esperando que el mensaje sea recibido por el receptor.
  - La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje ⇒ menos memoria.
- Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (los emisores se bloquean).
- Si bien send y sync_send son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de deadlock son mayores en comunicación sincrónica

Ejemplo

```
chan valores(int);

Process Productor
{  int datos[n];
   for [i=0 to n-1]
      {  #Hacer cálculos productor
         sync_send valores (datos[i]);
      }
}

Process Consumidor
{  int resultados[n];
   for [i=0 to n-1]
      {  receive valores (resultados[i]);
         #Hacer cálculos consumidor
      }
}
```

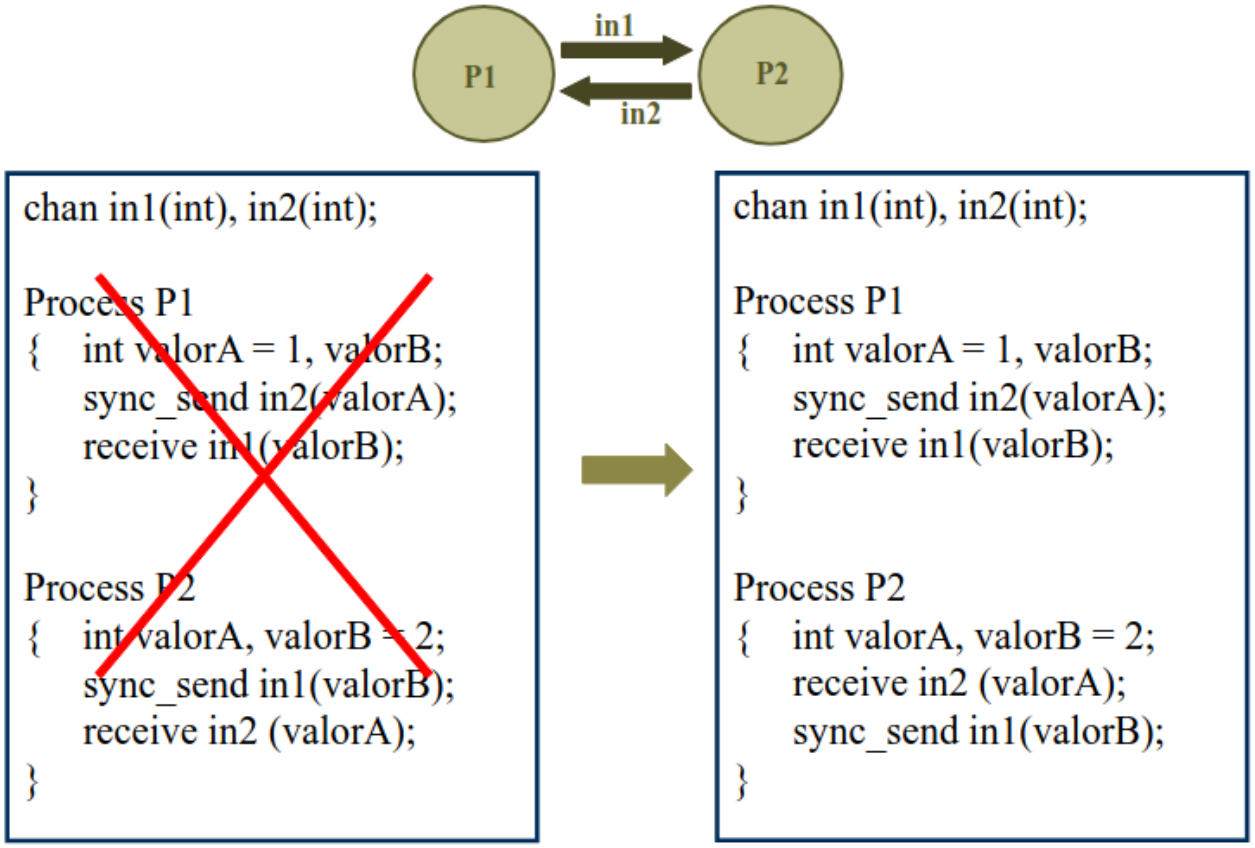


Comentarios

- Si los cálculos del productor se realizan mucho más rápido que los del consumidor en las primeras n/2 operaciones, y luego se realizan mucho más lento durante otras n/2 interacciones:
  - Con PMS los pares send/receive se completarán asumiendo la demora del proceso que más tiempo consuma. Si la relación de tiempo fuera 10 a 1 significaría multiplicar por 10 los tiempos totales.
  - Con PMA, al principio el productor es más rápido y sus mensajes se encolan. Luego el consumidor es más rápido y "descuenta" tiempo consumiendo la cola de mensajes.
- Mayor concurrencia en PMA. Para lograr el mismo efecto en PMS se debe interponer un proceso "buffer".
- ¿Que pasa si existe más de un productor/consumidor?.

Deadlock en PMS

Dos procesos que intercambian valores.



8

CSP - Lenguaje para PMS

- CSP (Communicating Sequential Processes, Hoare 1978) fue uno de los desarrollos fundamentales en Programación Concurrente. Muchos lenguajes reales (OCCAM, ADA, SR) se basan en CSP.
- Canal: link directo entre dos procesos en lugar de mailbox global. Son half- duplex y nominados.
- Las sentencias de Entrada (? o query) y Salida (! o shriek o bang) son el único medio por el cual los procesos se comunican. process A { . . . B ! e; . . . } process B { . . . A ? x; . . . }
- Para que se produzca la comunicación, deben matchear, y luego se ejecutan simultáneamente.
- Efecto: sentencia de asignación distribuida.

El lenguaje CSP (Hoare, 1978)

- Formas generales de las sentencias de comunicación:
  - Destino ! port(e1, ..., en);
  - Fuente ? port(x1, ..., xn);
- Destino y Fuente nombran un proceso simple, o un elemento de un arreglo de procesos. Fuente puede nombrar cualquier elemento de un arreglo (Fuente[*]).
- port son etiquetas que se usan para distinguir entre distintas clases de mensajes que un proceso podría recibir (puede omitirse si es sólo uno).
- Dos procesos se comunican cuando ejecutan sentencias de comunicación que hacen matching.

Comunicación Guardada

Limitaciones de ? y ! ya que son bloqueantes. Hay problema si un proceso quiere comunicarse con otros (quizás por != ports) sin conocer el orden en que los otros quieren hacerlo con él.
Las operaciones de comunicación (? y !) pueden ser guardadas, es decir hacer un AWAIT hasta que la confición sea verdadera.
El do e If de CSP usan los comandos guardados de Dijkstra(B -> S).

Las sentencias de comunicación guardada soportan comunicación no determinística:

- B; C -> S;
  - B puede omitirse y se asume true.
  - B y C forman la guarda.
  - La guarda tiene éxito si B es true y ejecutar C no causa demora.
  - La guarda falla si B es falsa.
  - La guarda se bloquea si B es true pero C no puede ejecutarse inmediatamente.
- Las sentencias de comunicación guardadas aparecen en if y do.
  - 

  - Ejecución:
    - Primero, se evalúan las guardas.
      - Si todas las guardas fallan, el if termina sin efecto.
      - Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
      - Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito.
    - Segundo, luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación de la guarda elegida.
    - Tercero, se ejecuta la sentencia S_i.
  - La ejecución del DO es similar (se repite hasta que todas las guardas fallen)

Clase 7

Conceptos Básicos

- El pasaje de Mensajes se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la comunicación unidireccional.
- Para resolver C/S la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas)
- Además, cada cliente necesita un canal de reply distinto...

> **Important**
> RPC (Remote Procedure Call) y Rendezvous ⇒ técnicas de comunicación y sincronización estre procesos que suponen un canal bidireccional ⇒ ideales para programar aplicaciones C/S

- RPC y Rendezvous combinan una interfaz "tipo monitor" con operaciones exportadas a través de llamadas externas (CALL) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados)

Diferencias entre RPC y Rendezvous

- Difieren en la manera de servir la invocación de operaciones
  - Un enfoque es declarar un procedure para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado (RPC porque el llamador y el cuerpo del procedure pueden estar en distintas máquinas). Para el cliente, durante la ejecución del servicio, es como si tuviera en un sitio el proceso remoto que lo sirve (Ej: JAVA).
  - El segundo enfoque es hacer rendezvous con un proceso existente. Un rendezvous es servido por una sentencia de entrada (o accept) que espera una invocación, la procesa y devuelve los resultados(Ej: Ada).

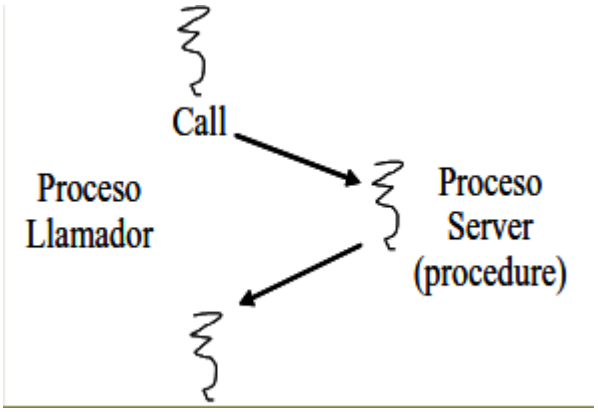
Remote Procedure Call (RPC)

- Los programas se descomponen en módulos (con procesos y procedures), que pueden residir en espacios de direcciones distintos.
- Los procesos de un módulo pueden compartir variables y llamar a procedures de ese módulo.
- Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.
- Los módulos tienen especificación e implementación de procedures

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

- Los procesos locales son llamados background para distinguirlos de las operaciones exportadas.
- Header de un procedure visible:
  op opname (formales) [returns result]
- El cuerpo de un procedure visible es contenido de una delcaración proc:
  proc opname(identif. formales) returns identificador resultado
  declaración de variables locales

sentencias
end

- El proceso (o procedure) en un módulo llama a un procedure en otro ejecutando:
  call Mname.opname (argumentos)
- Para un llamado local, el nombre del módulo se puede repetir
- La implementación de un llamado intermódulo es distinta que para uno local, ya que los dos módulos pueden estar en distintos espacios: un nuevo proceso sirve el llamado, y los argumentos son pasados como mensajes entre el llamador y el proceso server.
- El llamador se demora mientras el proceso servidor ejecuta el cuerpo del procedure que implementa opname.
- Cuando el server vuelve de opname envía los resultados al llamador y termina. Después de recibir los resultados, el llamador sigue.
- Si el proceso llamador y el procedure están en el mismo espacio de direcciones, es posible evitar crear un nuevo proceso.
- En general, un llamado será remoto ⇒ se debe crear un proceso server o alocarlo de un pool preexistente.
- 

Sincronización en módulos

Por sí mismo, RPC es solo un mecanismo de comunicación.

- Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador (como si éste estuviera ejecutando el llamado ⇒ la sincronización entre ambos es implícita).
- Necesitamos que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo). Esto comprende Exclusión Mutua y Sincronización por Condición.
- Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan:
  - Con exclusión mutua (un solo proceso por vez).
  - Concurrentemente.

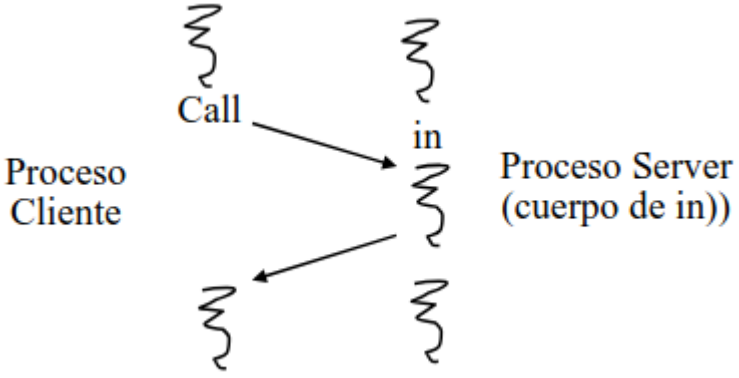
RPC en JAVA

Remote Method Invocation

- Java soporta le uso de RPC en programas distribuidos mediante la invocación de métodos remotos (RMI).
- Una aplicación que usa RMI tiene 3 componentes:
  - Una interfase que declara los headers para métodos remotos.
  - Una clase server que implementa la interfase.
  - Uno o más clientes que llaman a los métodos remotos.
- El server y los clientes pueden residir en máquinas diferentes.

Rendezvous

- RPC por si mismo sólo brinda un mecanismo de comunicación intermódulo. Dentro de un módulo es necesario programar la sincronización. Además, a veces son necesarios procesos extra sólo para manipular los datos comunicados por medio de RPC (ej: Merge).
- Rendezvous combina comunicación y sincronización:
  - Como con RPC, un proceso cliente invoca una operación por medio de un call, pero esta operación es servida por un proceso existente en lugar de por uno nuevo.
  - Un proceso servidor usa una sentencia de entrada para esperar por un call y actuar.
  - Las operaciones se atienden una por vez más que concurrentemente
- La especificación de un módulo contiene declaraciones de los headers de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones.
- Si un módulo exporta opname, el proceso server en el módulo realiza rendezvous con un llamador de opname ejecutando una sentencia de entrada:
  - in opname (parámetros formales) → S; ni
- Las partes entre in y ni se llaman operación guardada.
- Una sentencia de entrada demora al proceso server hasta que haya al menos un llamado pendiente de opname; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta S y finalmente retorna los parámetros de resultado al llamador.
  Luego, ambos procesos pueden continuar.
- A diferencia de RPC el server es un proceso activo.



- Combinando comunicación guardada con rendezvous:

in op_1 (formales$_1$) **and** B_1 **by** $e_1 \rightarrow S_1$;
□ ...
□ op_n (formales$_n$) **and** B_n **by** $e_n \rightarrow S_n$;
ni

- Los Bi son expresiones de sincronización opcionales.
- Los ei son expresiones de scheduling opcionales.
  - bi y ei pueden referenciar parametros formales

ADA - Lenguaje con Rendezvous

El lenguaje ADA

- Desarrollado por el Departamento de Defensa de USA para que sea el estandard en programación de aplicaciones de defensa.
- Desde el punto de vista de la concurrencia, un programa Ada tiene tasks (tareas) que pueden ejecutar independientemente y contienen primitivas de sincronización.
- Los puntos de invocación (entrada) a una tarea se denominan entrys y están especificados en la parte visible (header de la tarea).
- Una tarea puede decidir si acepta la comunicación con otro proceso, mediante la primitiva accept.
- Se puede declarar un type task, y luego crear instancias de procesos (tareas) identificado con dicho tipo (arreglo, puntero, instancia simple)

Tasks

La forma más común de especificación de task es:

```
Task nombre IS
declaraciones de ENTRYs
end;
```

La forma más común de cuerpo de task es:

```
TASK BODY nombre IS
declaraciones locales
BEGIN
sentencias
END nombre;
```

- Una especificación de TASK define una única tarea.
- Una instancia del correspondiente *task body* se crea en el bloque en el cual se declara la TASK.

Sincronización

Call: Entry call

- El rendezvous es el principal mecanismo de sincronziación en Ada y también es el mecanismo de comunicación primario
- Entry:
  - Declaración de entry simples y familia de entry (parámetros IN, OUT y IN OUT).
  - Entry call. La ejecución demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción). -> Tarea.entry(parámetros)
  - Entry call condicional:
    ```
    select entry call;
    sentencias adicionales;
    else
    sentencias;
    end select;
    ```
  - Entry call temporal:
    ```
    select entry call;
    sentencias adicionales;
    or delay tiempo
    sentencias
    end select;
    ```

Sentencia de Entrada Accept

- La tarea que declara un entry sirve llamados al entry accept:
  accept nombre (**parámetros formales**) **do** sentencias **end nombre**
- Demora la tarea hasta que haya una invocación, copia los parámetros reales en los parámetros formales, y ejecuta las sentencias. Cuando termina, los parámetros formales de salida son copiados a los parámetros reales. Luego ambos procesos continúan
- La **sentencia wait selectiva** soporta comunicación guardada

  select ***when*** $B_1 \Rightarrow$ ***accept*** E_1; sentencias$_1$
  or **...**
  or ***when*** $B_n \Rightarrow$ ***accept*** E_n; sentencias$_n$
  end select;

Clase 8

Paradigmas de interacción entre Procesos

- 3 esquemas básicos de interacción entre procesos: productor/consumidor, cliente/servidor e interacción entre pares.
- Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a otros paradigmas o modelos de interacción entre procesos

Resumen de los diferentes paradigmas

Master / Worker

Implementación distribuida del modelo Bag of Task.

algoritmos heartbeat

Los procesos periódicamente deben intercambiar información con mecanismos tipo send/recieve.

Algoritmos pipeline

La información recorre una serie de procesos utilizando alguna forma de recieve/send

probes (send) y echoes(receive)

La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.

algoritmos broadcast

Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas

token passing

En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas.

Servidores replicados

Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

Manager / Worker

- El concepto de bag of tasks usando variables compartidas supone que un conjunto de workers comparten una "bolsa" con tareas independientes. Los workers sacan una tarea de la bolsa, la ejecutan y posiblemente crean nuevas tareas que ponen en la bolsa (ejemplo en LINDA manejando un espacio compartido de tuplas).
- La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo en los workers.
- Analizaremos la implementación de este paradigma con mensajes en lugar de MC. Para esto un proceso manager implementará la "bolsa" manejando las tasks, comunicándose con los workers y detectando fin de tareas. Se trata de un esquema C/S
- Ejemplo: multiplicación de matrices ralas.

Heartbeat

- Paradigma heartbeat ⇒ útil para soluciones iterativas que se quieren paralelizar.
- Usando un esquema “divide & conquer” se distribuye la carga (datos) entre los workers; cada uno es responsable de actualizar una parte.
- Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos.

- Cada “paso” debiera significar un progreso hacia la solución.
- Formato general de los worker:

```
process worker [i =1 to numWorkers]
    { declaraciones e inicializaciones locales;
        while (no terminado)
        { send valores a los workers vecinos;
          receive valores de los workers vecinos;
          Actualizar valores locales;
        }
    }
```

- Ejemplo: grid computations (imágenes), autómatas celulares (simulación de fenómenos como incendios o crecimiento biológico).

Heartbeat - Topología de una red

Los procesadores están conectados por canales bidireccionales. Cada uno se comunica sólo con sus vecinos y conoce esos links.

- Modelización:
  - Procesador ⇒ proceso
  - Links de comunicación ⇒ canales compartidos
- Soluciones: los vecinos interactúan para intercambiar información local.

> **Algoritmo Heartbeat**
>
> Se expande enviando información, luego se contrae incorporando nueva información

- Procesos Nodo[p:1..n].
- Vecinos de p: vecinos[1:n] → vecinos[q] es true si q es vecino de p.
- Problema: computar top (matriz de adyacencia), donde top[p,q] es true si p y q son vecinos.

Cada nodo debe ejecutar n° de rondas para conocer la topología completa. Si el diámetro D de la red es conocido se resuelve con el siguiente algoritmo.

```
chan topologia[1:n] ([1:n,1:n] bool)

Process Nodo[p:1..n]
{  bool vecinos[1:n],  bool nuevatop[1:n,1:n],  top[1:n,1:n] = ([n*n] false);
   top[p,1..n] = vecinos;

   for (r =0 ; r < D; r++)
      { for [q = 1 to n st vecinos[q] ] send topologia[q](top);
        for [q = 1 to n st vecinos[q] ]
          {  receive topologia[p](nuevatop);
             top = top or nuevatop;
          }
      }
}
```

- Rara vez se conoce el valor de D.
- Excesivo intercambio de mensajes ⇒ los procesos cercanos al “centro” conocen la topología más pronto y no aprenden nada nuevo en los intercambios.
- El tema de la terminación ⇒ ¿local o distribuida?
- ¿Cómo se pueden solucionar estos problemas?
  - Después de r rondas, p conoce la topología a distancia r de él. Para cada nodo q dentro de la distancia r de p, los vecinos de q estarán almacenados en la fila q de top ⇒ p ejecutó las rondas suficientes tan pronto como cada fila de top tiene algún valor true.
  - Luego necesita ejecutar una última ronda para intercambiar la topología con sus vecinos.

- No siempre la terminación se puede determinar localmente.

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)

Process Nodo[p:1..n]
{  bool vecinos[1:n], activo[1:n] = vecinos, top[1:n,1:n] = ([n*n]false), nuevatop[1:n,1:n];
   bool qlisto, listo = false;
   int emisor;
   top[p,1..n] = vecinos;
   while (not listo)
     {   for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
         for [q = 1 to n st activo[q] ]
            {   receive topologia[p](emisor,qlisto,nuevatop);
               top = top or nuevatop;
               if (qlisto) activo[emisor] = false;
            }
          if (todas las filas de top tiene 1 entry true) listo=true;
     }
    for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
    for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop);
}
```

notas sobre cosas mencionadas en clase

Se pueden tomar en final o examen de promo:

- Matriz de procesos distribuida, cada uno tiene un número, hay que calcular el máximo en el sistema, conoce a una cantidad de vecinos específico, varía entre c/u la cantidad de iteraciones.
- Cantidad de interacciones es 2(n-1) por vecino, y si la vecindad es mas grande (8 en lugar de 2)la cantidad de iteraciones es n-1.
- Pueden preguntar como se podría mejorar, en el de la matriz, como el proceso sabe que distancia máxima tiene, deberia utilizarse la cantidad máxima de iteraciones qué ese debe hacer en base a donde está

Pipeline

- Un pipeline es un arreglo lineal de procesos "filtro" que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida.
- Estos procesos ("workers") pueden estar en procesadores que operan en paralelo, en un primer esquema a lazo abierto (W1 en el INPUT, Wn en el OUTPUT).
- Un segundo esquema es el pipeline circular, donde Wn se conecta con W1. Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe.
- En un tercer esquema posible (cerrado), existe un proceso coordinador que maneja la “realimentación” entre Wn y W1.
- Ejemplo: multiplicación de matrices en bloques.

Probe-Echo

- Las arquitecturas distribuidas se pueden asimilar a los nodos de grafos y árboles, con canales de comunicación que los vinculan.
- DFS es uno de los paradigmas secuenciales clásicos para visitar todos los nodos de un árbol o grafo. Este paradigma es análogo concurrente de DFS.
- Prueba-eco se basa en el envío de mensajes ("Probe") de un nodo al sucesor, y la espera posterior del mensaje de respuesta ("Echo").
- Los probes se envían en paralelo a todos los sucesores.
- Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (no se conoce) un número fijo de nodos activos (ejemplo: redes móviles).

Broadcast

- En la mayoría de las LAN cada procesador se conecta directamente con los otros. Estas redes normalmente soportan la primitiva broadcast:
  - broadcast ch(m);
- Los mensajes broadcast de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden.
- Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida. Ejemplo: semáforos distribuidos, la base es un ordenamiento total de eventos de comunicación mediante el uso de relojes lógicos.

Token Passing

- Un paradigma de interacción muy usado se basa en un tipo especial de mensaje (“token”) que puede usarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida. Un ejemplo del primer tipo de algoritmos es el caso de tener que controlar exclusión mutua distribuida.
- Aunque el problema de la SC se da principalmente en programas de MC, puede encontrarse en programas distribuidos cuando hay algún recurso compartido que puede usar un único proceso a la vez. Generalmente es una componente de un problema más grande, tal como asegurar consistencia en un sistema de BD.
- Soluciones posibles: Monitor activo que da permiso de acceso (ej: locks en archivos), semáforos distribuidos (usando broadcast, con gran intercambio de mensajes), o token ring (descentralizado y fair).

Ejemplo solución

- Ejemplo de solución al problema de la SC en un ámbito distribuido con **Token Ring** (**Token Passing** en forma de anillo).
- Por cada proceso **worker** se agrega un proceso **auxiliar**.
  - El **TOKEN** viaja entre los auxiliares conectados en forma de anillo.

- El worker solo se comunica con su proceso auxiliar cuando quiere acceder a la SC. Y espera a que este le indique que puede usarla. Cuando el worker termina de usar la SC le avisa a su proceso auxiliar que terminó de usar la SC.
- Cuando un proceso auxiliar recibe el TOKEN, si su **worker** quiere usar la SC le avisa que ya lo puede hacer y espera a que le avise que la liberó. Y pasa el token al siguiente auxiliar del anillo.

Servidores Replicados

Un server puede ser replicado cuando hay múltiples instancias de un recurso: cada server maneja una instancia.

- La replicación también puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.
- Ejemplo: problema de los filósofos
  - Modelo centralizado: los Filósofo se comunican con UN proceso Mozo que decide el acceso o no a los recursos.
  - Modelo distribuido: supone 5 procesos Mozo, cada uno manejando un tenedor.'Un Filósofo puede comunicarse con 2 Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos NO se comunican entre ellos.
  - Modelo descentralizada: cada Filósofo ve un único Mozo. Los Mozos se comunican entre ellos (cada uno con sus 2 vecinos) para decidir el manejo del recurso asociado a “su” Filósofo.

Clase 9

Librerías para memoria compartida

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)

- Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones "multithreading".
- En principio estos mecanismos fueron heterogéneos y poco portables. A mediados de los 90 la org POSIX auspició el desarrollo de una biblioteca en C para multithreading (Pthreads).
- Pthreads es una biblioteca para programación paralela en **memoria compartida**, se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

POSIX - Api de Threads

- Numerosas APIs para el manejo de Threads.
- Normalmente llamada Pthreads, POSIX a emergido como un API estandard para manejo de Threads, provista por la mayoría de los vendedores.
- Los conceptos que se discutirán son independientes de la API y pueden ser igualmente válidos para utilizar JAVA Threads, NT Threads, Solaris Threads, etc.
- Funciones reentrantes.

Creación y Terminación

- Pthreads provee funciones básicas para especificar concurrencia:

```
#include <pthread.h>
int pthread_create (pthread_t *thread_handle, const pthread_attr_t
*attribute,
void * (*thread_function)(void *), void *arg);
int pthread_exit (void *res);
int pthread_join (pthread_t thread, void **ptr);
int pthread_cancel (pthread_t thread);
```

- El "main" debe esperar a que todos los threads terminen.

Primitivas de Sincronización

Exclusión mutua

- Las secciones críticas se implementan en Pthreads utilizando mutex locks (bloqueo por exclusión mutua) por medio de variables mutex.
- Una variable mutex tienen dos estados: locked (bloqueado) and unlocked (desbloqueado). En cualquier instante, sólo UN thread puede bloquear un mutex. Lock es una operación atómica.
- Para entrar en la sección crítica un Thread debe lograr tener control del mutex (bloquearlo).
- Cuando un Thread sale de la SC debe desbloquear el mutex.
- Todos los mutex deben inicializarse como desbloqueados.
- La API Pthreads provee las siguientes funciones para manejar los mutex:
- ```
  int pthread_mutex_lock ( pthread_mutex_t *mutex);
  int pthread_mutex_unlock (pthread_mutex_t *mutex);
  int pthread_mutex_init (pthread_mutex_t *mutex,
  const pthread_mutexattr_t *lock_attr);
  ```

Productores y consumidores con Exclusión mutua

El escenario de productores-consumidores impone las siguientes restricciones:
  ```

- Un thread productor no debe sobrescribir el buffer compartido cuando el elemento anterior no ha sido tomado por un thread consumidor.
- Un thread consumidor no puede tomar nada de la estructura compartida hasta no estar seguro de que se ha producido algo anteriormente.
- Los consumidores deben excluirse entre sí.
- Los productores deben excluirse entre sí.
- En este ejemplo el buffer es de tamaño 1

Tipos de exclusión mutua (MUTEX)

- Pthreads soporta tres tipos de Mutexs (Locks): Normal, Recursive y Error check
  - Un mutex con el atributo Normal no permite que un thread que lo tiene bloqueado vuelva a hacer un lock sobre él (deadlock).
  - Un Mutex con el atributo Recursive SI permite que un thread que lo tiene bloqueado vuelva a hacer un lock sobre él. Simplemente incrementa una cuenta de control.
  - Un Mutex con el atributo ErrorCheck responde con un reporte de error al intento de un segundo bloqueo por el mismo thread.
- El tipo de Mutex puede setearse entre los atributos antes de su inicialización.

Overhead de Bloqueos por Exclusión Mutua

Los locks representan puntos de serialización → si dentro de las secciones críticas ponemos segmentos largos de programa tendremos una degradación importante de perfomance.

- A menudo se puede reducir el overhead por espera ociosa, utilizando la función pthread_mutex_trylock. Retorna el control informando si pudo hacer o no el lock.
  - int pthread_mutex_trylock (pthread_mutex_t *mutex_lock).
  - Evita tiempos ociosos.
  - Menos costoso por no tener que manejar las colas de espera.

Variables Condición

- Podemos utilizar variables de condición para que un thread se autobloquee hasta que se alcance un estado determinado del programa.
- Cada variable de condición estará asociada con un predicado. Cuando el predicado se convierte en verdadero (TRUE) la variable de condición da una señal para el/los threads que están esperando por el cambio de estado de la condición.
- Una única variable de condición puede asociarse a varios predicados (difícil el debug).
- Una variable de condición siempre tiene un mutex asociada a ella. Cada thread bloquea este mutex y testea el predicado definido sobre la variable compartida.
- Si el predicado es falso, el thread espera en la variable condición utilizando la función pthread_cond_wait (NO USA CPU).

La API Pthreads provee las siguientes funciones para manejar
las variables condición:

```
int pthread_cond_wait ( pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_timedwait ( pthread_cond_t *cond, pthread_mutex_t *mutex
const struct timespec *abstime)
int pthread_cond_signal (pthread_cond_t *cond)
int pthread_cond_broadcast (pthread_cond_t *cond)
int pthread_cond_init ( pthread_cond_t *cond, const pthread_condattr_t
*attr)
int pthread_cond_destroy (pthread_cond_t *cond)
```

Atributos y sincronización

La Api pthreads permite que se pueda cambiar los atributos por defecto de las entidades, utilizando attributes objects.
Un attribute object es una estructura de datos que describe las propiedades de la entidad en cuestión (thread, mutex, variable de condición). Una vez que estas propiedades están establecidas, el attribute object es pasado al método que inicializa la entidad.
Ventajas:

- Esta posibilidad mejora la modularidad
- Facilidad de modificación del código

Atributos para Threads

- La API Pthreads provee las siguientes funciones para manejar los atributos para Threads:

```
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);
```

- Las propiedades asociadas con el attribute object pueden ser cambiadas con las siguientes f
- Funciones:

```
pthread_attr_setdetachstate
pthread_attr_setguardsize_np
pthread_attr_setstacksize
pthread_attr_setinheritsched
pthread_attr_setschedpolicy
pthread_attr_setschedparam
```

Atributos para Mutex

- La API Pthreads provee las siguientes funciones para manejar los atributos para Mutex:

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_settype_np ( pthread_mutexattr_t *attr, int type);
```

- Aquí type especifica el tipo de mutex y puede tomar los valores:
  - PTHREAD_MUTEX_NORMAL_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_N

Semáforos en Pthreads

- Los threads pueden sincronizar por semáforos (librería semaphore.h).
- Declaración y operaciones con semáforos en Pthreads:
  - sem_t semaforo → se declaran globales a los threads.
  - sem_init (&semaforo, alcance, inicial) → en esta operación se inicializa el semáforo semaforo. Inicial es el valor con que se inicializa el semáforo. Alcance indica si es compartido por los hilos de un único proceso (0) o por los de todos los procesos (≠ 0).
  - sem_wait(&semaforo) → equivale al P.
  - sem_post(&semaforo) → equivale al V.
  - Existen funciones extras para: wait condicional, obtener el valor de un semáforo y destruir un semáforo (ESTE TIPO DE FUNCIONES EXTRAS NO SE PUEDEN USAR EN LA PRÁCTICA DE LA MATERIA).

Monitores en Pthreads

- Pthreads no permite manejar la Exclusión Mutua por medio de las variables mutex.
- Pthreads nos permite manejar la Sincronización por Condición utilizando variables condición para que un thread se auto bloquee hasta que se alcance un estado determinado del programa. Una variable de condición siempre tiene un mutex asociada a ella.
- Pthreads no posee “Monitores”, pero con las dos herramientas que mencionamos se puede similar el uso de monitores: con mutex se hace la exclusión mutua que nos brindaba implicitamente el monitor, y con las variables condición la sincronización.
  - El acceso exclusive al monitor se simula usando una variable mutex la cual se bloquea antes del llamada al procedure y se desbloquea al terminar el mismo (una variable mutex diferente para cada monitor).
  - Cada llamado de un proceso a un procedure de un monitor debe ser reemplazado por el código de ese procedure.

Librerías para manejo de PM

Operaciones Send y Recieve

- Los prototipos de las operaciones son:
  - Send (void *sendbuf, int nelems, int dest)
  - Receive (void *recvbuf, int nelems, int source)
- Diferentes protocolos para Send y Receive.

Send y recieve bloqueante

- Para asegurar la semántica del SEND → no devolver el control del Send hasta que el dato a trasmitir esté seguro (Send bloqueante).
- Ociosidad del proceso.
- Hay dos posibilidades:
  - Send/Receive bloqueantes sin buffering.
  - Send/Receive bloqueantes con buffering.

Send y recieve no bloqueante

- Para evitar overhead (ociosidad o manejo de buffer) se devuelve el control de la operación inmediatamente.
- Requiere un posterior chequeo para asegurarse la finalización de la comunicación.
- Deja en manos del programador asegurar la semántica del SEND.
- Hay dos posibilidades:
  - Send/Receive no bloqueantes sin buffering.
  - Send/Receive no bloqueantes con buffering

Message Passing Interface (MPI)

Librería MPI

- Existen numerosas librerías de pasaje de mensajes (no compatibles).
- MPI define una librería estándar que peude ser empleada desde C o Fortran (y potencialmente desde otros lenguajes).
- El estándar MPI define la sintaxis y la semántica de más de 125 rutinas.
- Hay implementaciones de MPI de la mayoría de los proveedores de hardware.
- Modelo SPMD.
- Todas las rutinas, tipos de datos y constantes de MPI tienen el prefijo "MPI_". El código de retorno para operaciones terminadas exitosamente es MPI_SUCCESS.
- Básicamente con 6 rutinas podemos escribir programas paralelos basados en pasaje de mensajes: MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send y MPI_Recv.

Inicio y finalización de MPI

- MPI_Init: se invoca en todos los procesos antes que cualquier otro llamado a rutinas MPI. Sirve para inicializar el entorno MPI.
  - MPI_Init (int *argc, char **argv)
  - Algunas implementaciones de MPI requieren argc y argv para inicializar el entorno
- MPI_Finalize: se invoca en todos los procesos como último llamado a rutinas MPI. Sirve para cerrar el entorno MPI.
  - MPI_Finalize ()

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 que procesos pertenecen a él.
- En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar.

Adquisición de Información

- MPI_Comm_size: indica la cantidad de procesos en el comunicador.
  MPI_Comm_size (MPI_Comm comunicador, int *cantidad).
- MPI_Comm_rank: indica el "rank" (identificador) del proceso dentro de ese comunicador.
  - MPI_Comm_rank (MPI_Comm comunicador, int * rank)
    - rank es un valor entre [0..CANTIDAD]
    - Cada proceso puede tener un rank diferente en cada comunicador

Tipos de Datos para las comunicaciones

Tipo de Datos MPI	Tipo de Datos C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Comunicación punto a punto

- Diferentes protocolos para send.
  - Send bloqueantes con buffering (Bsend).
  - Send bloqueantes sin buffering (Ssend).
  - Send no bloqueantes (Isend).
- Diferentes protocolos para Recv.

    - Recv bloqueantes (Recv).
    - Recv no bloqueantes (Irecv)

Comunicación bloqueante punto a punto

- MPI_Send, MPI_Ssend, MPI_Bsend: rutina básica para enviar datos a otro proceso.
    - MPI_Send (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador)
    - Valor de tag entre [0..MPI_TAG_UB].
- MPI_Recv: rutina básica para recibir datos de otro proceso
    - MPI_Recv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador, MPI_Status* estado)
    - Comodines MPI*ANY_SOURCE y MPI_ANY_TAG.*
    - MPI_Get_count para obtener la cantidad de elementos recibido.
        - MPI_Get_count(MPI_Status *estado, MPI_Datatype tipoDato, int* cantidad)

Comunicación no bloqueante punto a punto

- Comienzan la operación de comunicación e inmediatamente devuelven el control (no se asegura que la comunicación finalice correctamente).
    - MPI_Isend (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador, MPI_Request* solicitud)
    - MPI_Irecv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador, MPI_Request* solicitud)
- MPI_Test: testea si la operación de comunicación finalizó.
- MPI_Wait: bloquea al proceso hasta que finaliza la operación.
- Este tipo de comunicación permite solapar computo con comunicación. Evita overhead de manejo de buffer. Deja en manos del programador asegurar que se realice la comunicación correctamente.

Consulta de mensajes pendientes

- Información de un mensaje antes de hacer el Recv (Origen, Cantidad de elementos, Tag).
- MPI_Probe: bloquea el proceso hasta que llegue un mensaje que cumpla con el origen y el tag.
- MPI_Iprobe: cheqea por el arribo de un mensaje que cumpla con el origen y tag.

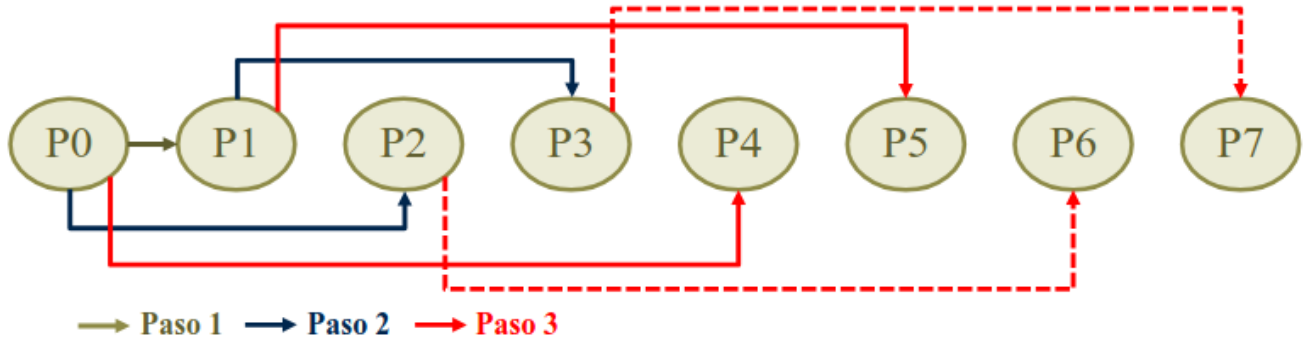
Comunicaciones Colectivas

MPI provee un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociado con un comunicador. Todos los procesos del comunicador deben llamar a la rutina colectiva:

- MPI_Barrier
- MPI_Bcast
- MPI_Scatter - MPI_Scatterv

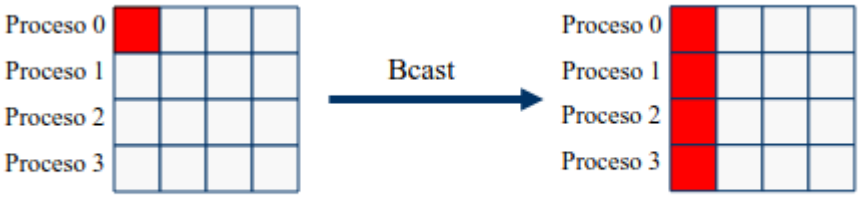
- MPI_Gather - MPI_Gatherv
- MPI_Reduce

Ventajas del uso de comunicaciones colectivas.

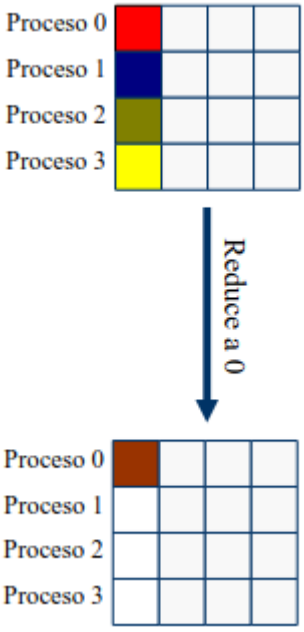


54

- Sincronización en una barrera.
  - MPI_Barrier(MPI_Comm comunicador)
- Broadcast: un proceso envía el mismo mensaje a todos los otros procesos (incluso a él) del comunicador.



- Reducción de todos a uno: combina los elementos enviados por cada uno de los procesos (inclusive el destino) aplicando una cierta operación.



- Gather: recolecta el vector de datos de todos los procesos (inclusive el destino) y los concatena en orden para dejar el resultado en un único proceso.

Minimizando los overheads de comunicación

- Maximizar la localidad de datos.
- Minimizar el volumen de intercambio de datos.
- Minimizar la cantidad de comunicaciones.
- Considerar el costo de cada bloque de datos intercambiado.
- Replicar datos cuando sea conveniente.
- Lograr el overlapping de cómputo (procesamiento) y comunicaciones.
- En lo posible usar comunicaciones asincrónicas.
- Usar comunicaciones colectivas en lugar de punto a punto

Clase 10

Arquitecturas Paralelas

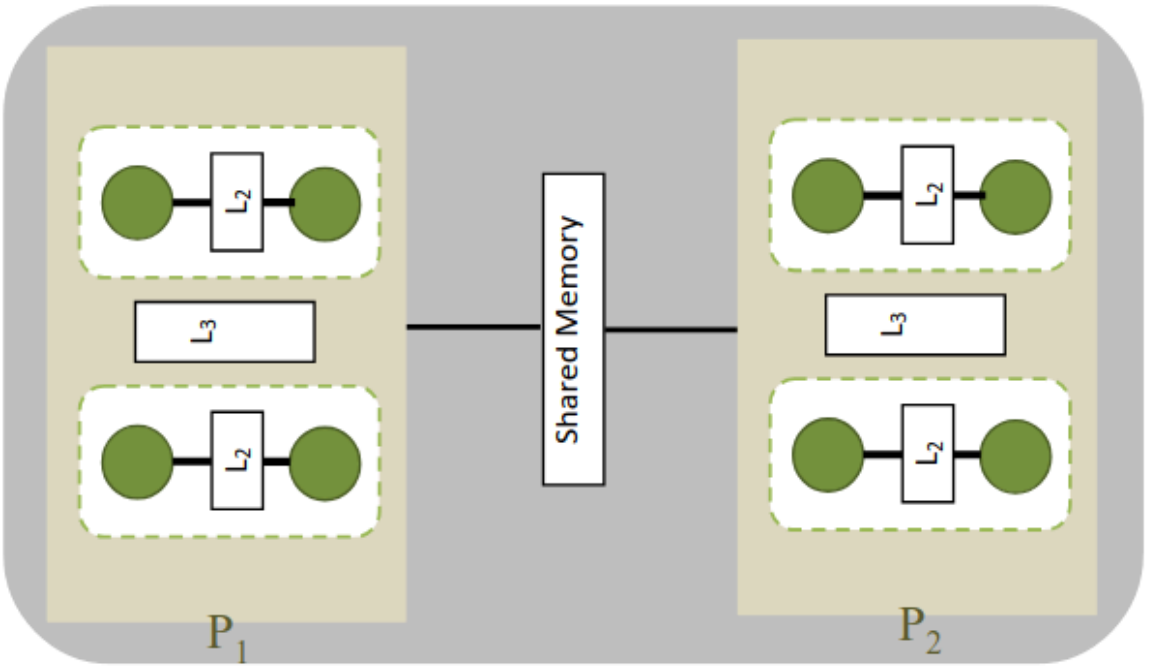
Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

- Por la organización del espacio de direcciones.
- Por la granularidad.
- Por el mecanismo de control.
- Por la red de interconexión.

Clasificación por el Espacio de Direcciones

- Las arquitecturas paralelas se clasifican según su espacio de direcciones en:
  - Memoria Compartida.
  - Memoria Distribuida.
- Esta clasificación se relaciona con el modelo de comunicación a utilizar:
  - Accesos a Memoria Compartida (memoria compartida).
  - Intercambio de mensajes (principalmente memoria distribuida).
- En algunos casos también tenemos en la misma plataforma ambos mecanismos.
- Multiprocesadores de memoria compartida.
  - Interacción modificando datos en la memoria compartida.
  - Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problemas de sincronización y consistencia.
  - Esquemas NUMA para mayor número de procesadores distribuidos.
  - Problema de consistencia.
- Ejemplo de multiprocesador de memoria compartida: multicore de 8 núcleos.



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

Clasificación por mecanismo de control

Se basa en la manera en que las instrucciones son ejecutadas sobre los datos.
Clasifica las arquitecturas en 4 clases:

- SISD (Single Instruction Single Data).
- SIMD (Single Instruction Multiple Data).
- MISD (Multiple Instruction Single Data).
- MIMD (Multiple Instruction Multiple Data).

SISD: Single Instruction Single Data

- Instrucciones ejecutadas en secuencia, una por ciclo de instrucción.
- La memoria afectada es usada sólo por ésta instrucción.
- Usada por la mayoría de los uní procesadores.
- 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.

SIMD: Single Instruction Multiple Data

- Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos.
- Los procesadores en general son muy simples.
- El host hace broadcast de la instrucción. Ejecución sincrónica y determinística.
- Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones.
- Adecuados para aplicaciones con alto grado de regularidad, (por ejemplo procesamiento de imágenes).

MIMD: Multiple Instruction Multiple Data

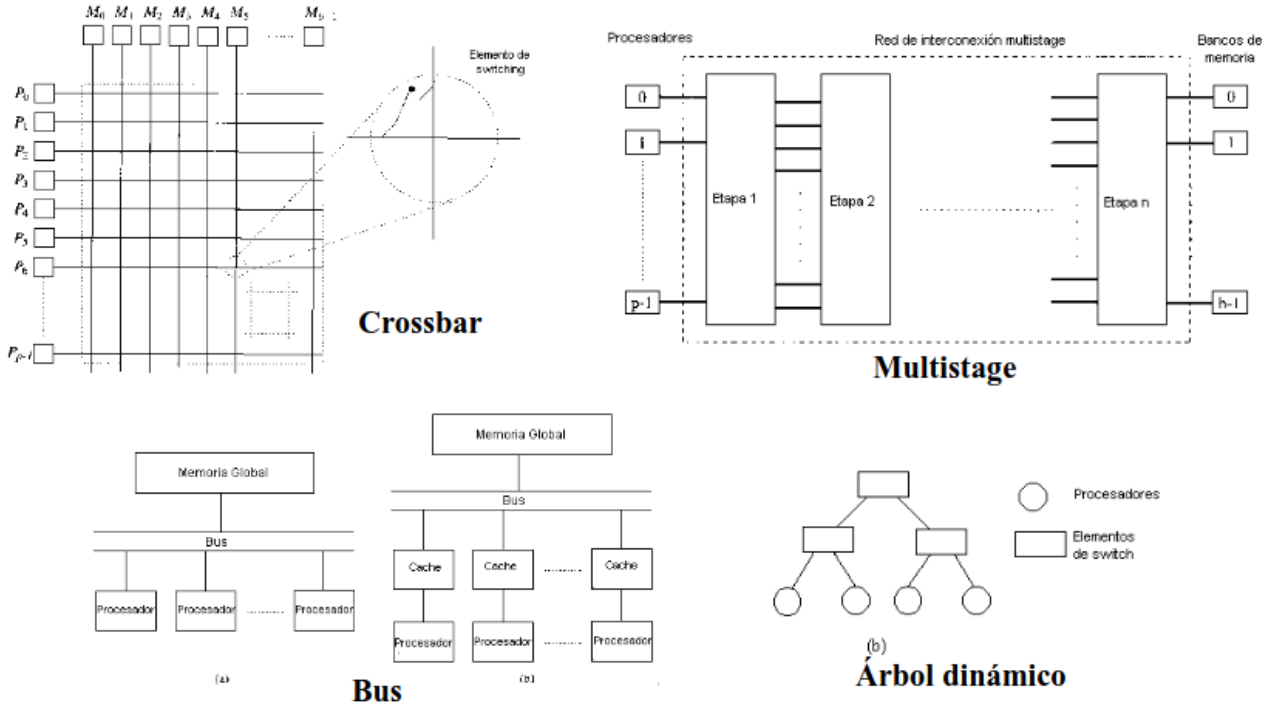
- Cada procesador tiene su propio flujo de instrucciones y de datos cada uno ejecuta su propio “programa” a su ritmo.
- Pueden ser con memoria compartida o distribuida.
- Sub-clasificación de MIMD:
- MPMD (multiple program multiple data): cada procesador ejecuta su propio programa (ejemplo con PVM).
- SPMD (single program multiple data): hay un único programa fuente y cada procesador ejecuta su copia independientemente (ejemplo con MPI).

Clasificación por la Red de interconexión

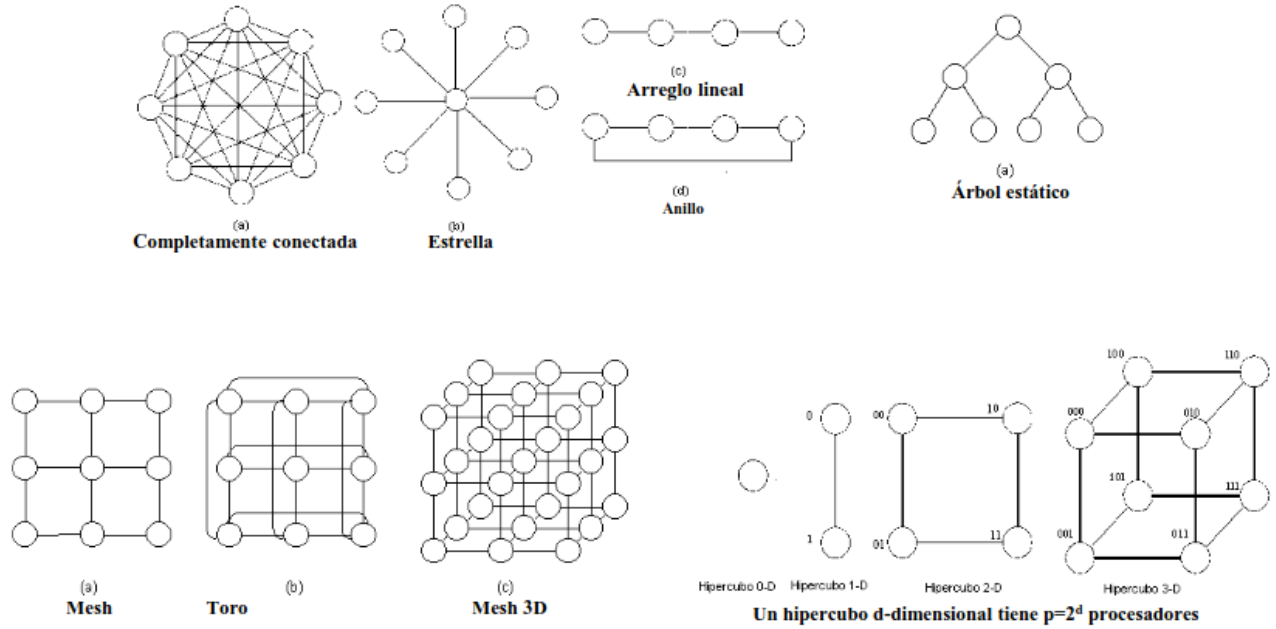
Tanto en memoria compartida como en pasaje de mensajes las máquinas pueden construirse conectando procesadores y memorias usando diversas redes de interconexión:

- Las redes estáticas constan de links punto a punto. Típicamente se usan para máquinas de pasaje de mensajes.
- Las redes dinámicas están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de memoria compartida.
  El diseño de la red de interconexión depende de una serie de factores (ancho de banda, tiempo de startup, paths estáticos o dinámicos, operación sincrónica o asincrónica, topología, costo, etc.).

Redes de interconexión Dinámicas



Redes de interconexión estáticas



Diseño de Algoritmos Paralelos

- La mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes.
- Puede darse un enfoque metódico para maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas, y reducir el costo de backtracking por malas elecciones.
- Aspectos independientes de la máquina tales como la concurrencia son considerados tempranamente, y los aspectos específicos de la máquina se demoran.

¿Por qué es compleja la programación paralela?

- Decidir cuál es la granularidad óptima de las tareas.
- Mapear tareas y datos a los nodos físicos de procesamiento (¿en forma estática o dinámica?)
- Manejar comunicación y sincronización.
- Asegurar corrección. Evitar deadlocks. Evitar desbalances.
- Obtener un cierto grado de Tolerancia a Fallos.
- Manejar la heterogeneidad.
- Lograr escalabilidad en todos los casos (potencia, tamaño de la arquitectura y del problema).
- Consumo energético.

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, arquitectura de destino, o temas de IS pueden llevar a descartar algunas posibilidades detectadas en esta etapa, revisando la partición original 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 asociarle 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.
- Descomposición funcional: primero descompone el cómputo en tareas disjuntas y luego trata los datos.

- Los requerimientos de datos pueden ser disjuntos (partición completa) o superponerse significativamente (necesidad de comunicación para evitar replicación de datos). En el segundo caso, probablemente convenga descomponer el dominio.
- Inicialmente se busca no replicar cómputo y datos. Esto puede revisarse luego para reducir costos.
- 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

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: 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 tareas 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 ≠ procesadores para mejorar la concurrencia o poner tareas que se comunican con frecuencia en = procesador 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 condicionará 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 de Rendimiento

- En el mundo serial la performance con frecuencia es medida teniendo en cuenta los requerimientos de tiempo y memoria de un programa.
- En un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance.
- Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución.
- A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores (sistema paralelo = algoritmo + arquitectura sobre la que se implementa).
- La diversidad torna complejo el analisis de performance
- En la medición de performance es usual elegir un problema y testear el tiempo variando el número de procesadores. Aquí subyacen las nociones de speedup y eficiencia, y la ley de Amdahl.
- Otro tema de interés es la escalabilidad, que da una medida de usar eficientemente un número creciente de procesadores.

Speedup (S)

- S es el cociente entre el tiempo de ejecución del algoritmo serial conocido más rápido (T_S) y el tiempo de ejecución paralelo del algoritmo elegito (T_p):
  - $S = T_S/T_p$

- Speedup óptimo depende de la arquitectura (en homogénea P).

$$S_{\acute{o}ptimo} = \sum_{i=0}^{P} \frac{PotenciaCalculo(i)}{PotenciaCalculo(mejor)}$$

- Rango de valores: en general entre 0 y S_{optimo}.
- Speedup lineal o perfecto, sublineal y superlineal.

Eficiencia (E)

- Cociente entre Speedup y Speedup Óptimo
  - $E = S/S_{optimo}$
- Mide la fracción de tiempo en que los procesadores son útiles para el cómputo.
- El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.

Escalabilidad de los sistemas paralelos

- Es muy difícil extrapolar la perfomance de un sistema paralelo, a partir de configuraciones con pocos procesadores y conjuntos de datos reducidos.
  - No sirven los estudios con 2, 4, 8 procesadores que proyectan el S_p alcanzable con 128 o 256 procesadores o el tiempo de procesamiento cuando tengamos 100 o 1000 veces más datos… ¿Por qué?
  - Básicamente porque los resultados con pequeños conjuntos de datos están afectados por la localidad en el manejo de la memoria, y los resultados con pocos procesadores porque las comunicaciones no computan los costos relacionados con la distancia entre procesadores y la disminución del ancho de banda efectivo.

Factores que limitan el speedup

- Alto porcentaje de código secuencial (Ley de Amdahl).
- Alto porcentaje de entrada/salida respecto de la computación.
- Algoritmo no adecuado (necesidad de rediseñar).
- Excesiva contención de memoria (rediseñar código para localidad de datos).
- Tamaño del problema (puede ser chico, o fijo y no crecer con p).
- Desbalance de carga (produciendo esperas ociosas en algunos procesadores).
- Overhead paralelo: ciclos adicionales de CPU para crear procesos, sincronizar, etc.

Paradigmas de Programación Paralela

- Paradigma de programación: clase de algoritmos que resuelve distintos problemas, pero tienen la misma estructura de control.
- Para cada paradigma puede escribirse un esqueleto algoritmico que define la estructura de control común.
- Dentro de la programación paralela pueden encontrarse paradigmas que permiten encuadrar los problemas en alguno de ellos.

- En cada paradigma, los patrones de comunicación son muy similares en todos los casos.

Cliente / Servidor

- 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. Comunicación bidireccional. Atención de a un cliente a la vez, o a varios con multithreading.
- Mecanismo de invocación variados (rendezvous, RPC, monitores).
- El soporte distribuido puede ser simple (LAN) o extendido a la web

Master/slave o master/worker

- Basado en organizaciones del mundo real.
- El master envía iterativamente datos a los workers y recibe resultados de éstos.
- Posible "cuello de botella" (por ejemplo, por tareas muy chicas o slaves muy rápidos) → elección del grano adecuado.
- Dos casos de acuerdo a las dependencias de las iteraciones:
  - Iteraciones dependientes: el master necesita los resultados de todos los workers para generar un nuevo conjunto de datos.
  - Entradas de datos independientes: los datos llegan al maestro, que no necesita resultados anteriores para generar un nuevo conjunto de datos
- Dos opciones para la distribución de los datos:
  - Distribuir todos los disponibles, de acuerdo a alguna política (estático).
  - Bajo petición o demanda (dinámico).
- Existen variantes, pero básicamente un procesador es responsable de la coordinación y los otros de resolver los problemas asignados.
- Es una variación de SPMD donde hay dos programas en lugar de sólo uno.
- Casos:
  - Procesadores heterogéneos y con distintas velocidades → problemas con el balance de carga.
  - Trabajo que debe realizarse en "fases" → sincronización.
  - Generalización a modelo multi-nivel o jerárquico

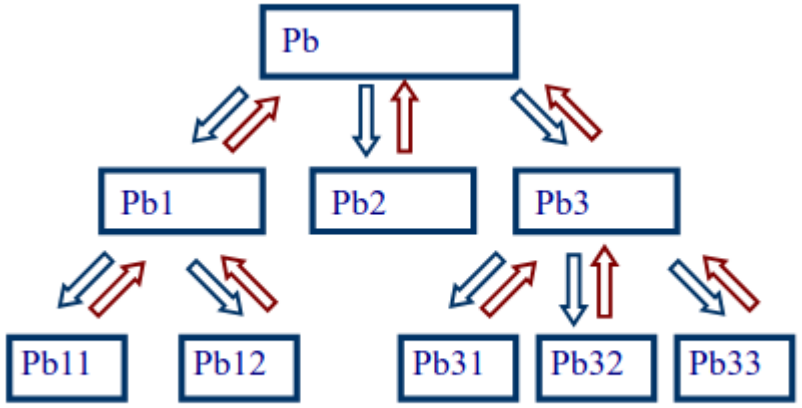
Pipeline y Algoritmos Sistólicos

- El problema se particiona en una secuencia de pasos. El stream de datos pasa entre los procesos, y cada uno realiza una tarea sobre él.
- Ejemplo: filtrado, etiquetado y análisis de escena en imágenes.
- Mapeo natural a un arreglo lineal de procesadores.
- Extensiones:
  - Procesadores especializados no iguales.
  - Más de un procesador para una tarea determinada.

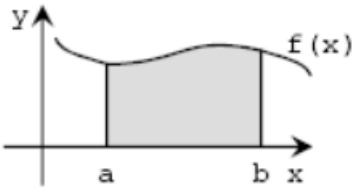
- El flujo puede no ser una línea simple (ejemplo: ensamble de autos con varias líneas que son combinadas) -> procesamiento sistólico.

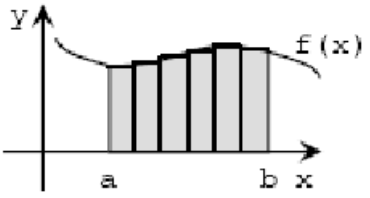
Dividir y Conquistar

- En general implica paralelismo recursivo donde el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (dividir y conquistar).
- División repetida de problemas y datos en subproblemas más chicos (fase de dividir); resolución independiente de éstos (conquistar), con frecuencia de manera recursiva. Las soluciones son combinadas en la solución global (fase de combinar).
- La subdivisión puede corresponderse con la descomposición entre procesadores. Cada subproblema puede mapearse a un procesador. Cada proceso recibe una fracción de datos: si puede los procesa; sino, crea un n° de "hijos" y les distribuye los datos.
- Ejemplos clásicos son el “sorting by merging”, el cálculo de raíces en funciones continuas, problema del viajante.

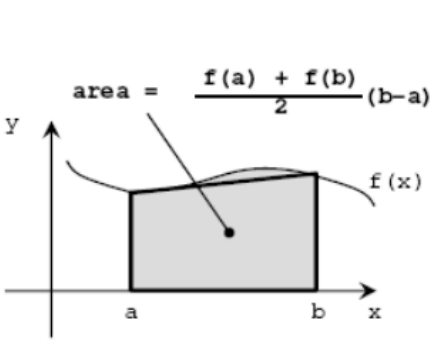


- Ejemplo el “Problema de la cuadratura”: calcular una aprox. de la integral de una función continua f(x) en el intervalo de a a b

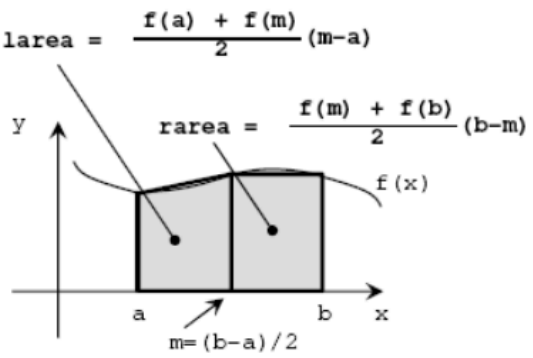




- Procesamiento recursivo adaptivo



(a) First approximation (area)



(b) Second approximation (larea + rarea)

SPMD

El programador genera un programa único que ejecuta cada nodo sobre una porción del dominio de datos. La diferente evaluación de un predicado en sentencias condicionales permite que cada nodo tome distintos caminos del programa.
Dos fases: 1) elección de la distribución de datos y 2) generación del programa paralelo.

1. Determinar el lugar que ocuparán los datos en los nodos. La carga es proporcional al número de datos asignado a cada nodo. Dificultades en computaciones irregulares y máquinas heterogéneas.
2. Convierte al programa secuencial en SPMD. En la mayoría de los lenguajes, depende de la distribución de datos.
   Suele implicar paralelismo iterativo donde un programa consta de un conjunto de procesos los cuales tiene 1 o más loops. Cada proceso es un programa iterativo.
   Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones.