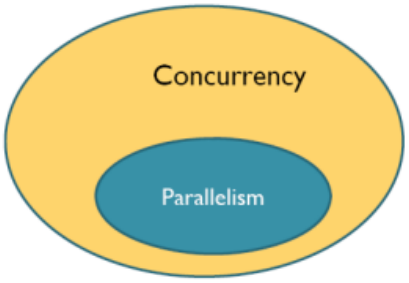
PROGRAMACIÓN CONCURRENTE

Clase 1 | *INTRODUCCIÓN*

CONCURRENCIA: Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores, que aporta la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente.

PARALELISMO: Se encuentra incluido dentro de la concurrencia y se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.



Clase 1

La concurrencia tiene una ejecución **no deterministica**, es decir, que ejecuciones con la misma “entrada” puede generar diferentes “salidas”.
Es necesaria por:

. *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.

Objetivos:

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

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

POSIBLES COMPORTAMIENTOS DE LOS PROCESOS

Un único hilo o flujo de control

→ programación secuencial, monoprocesador.

Múltiples hilos o flujos de control

→ programa concurrente: *Un programa concurrente especifica dos o más "programas secuenciales" que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos. Un mismo procesador realiza ambas tareas intercalandolas, trabajando sobre ambas "al mismo tiempo".*

→ programas paralelos: *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, siendo esto una solución paralela.*
Disponemos de múltiples procesadores, cada uno abocado a una tarea.

Los procesos cooperan y compiten…

→ Competencia: Los distintos procesos compiten por un único recurso compartido. Puede generar *deadlock* e *inanición*.

→ Cooperación: Los procesos se combinan para resolver una tarea común, deben *sincronizarse*.

PROCESOS E HILOS

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

. **Procesos livianos, threads o hilos:**

• Un proceso "liviano" 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).

COMUNICACIÓN ENTRE PROCESOS

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

• Por Memoria Compartida: Los procesos intercambian información sobre una única memoria compartida o actúan coordinadamente sobre datos residentes en ella. Estos no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a bloquear y liberar el acceso a la misma. La solución más elemental es una variable de control tipo *"semáforo"* que habilite o no el acceso de un proceso a la memoria compartida.

• Por Pasaje de Mensajes: Es necesario establecer un canal (lógico o físico) para transmitir información entre procesos. También el lenguaje debe proveer un protocolo adecuado. Para que la comunicación sea efectiva los procesos deben "saber" cuándo tienen mensajes para leer y cuando deben transmitir mensajes.

SINCRONIZACIÓN ENTRE PROCESOS

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

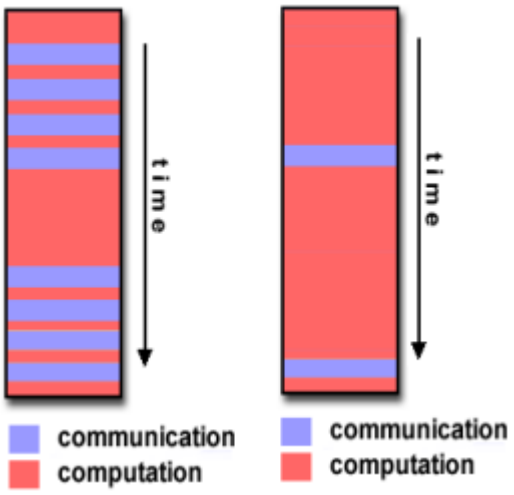
- Por exclusión mutua: Asegura que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo. Si el programa tiene secciones críticas que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.
- Por condición: Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

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

PRIORIDAD Y GRANULARIDAD

. Un proceso que tiene mayor **prioridad** puede causar la suspensión (preemption) de otro proceso concurrente. Análogamente puede tomar un recurso compartido, obligando a retirarse a otro proceso que lo tenga en un instante dado.

. La **granularidad** de una aplicación está dada por la relación entre el cómputo y la comunicación.



Grano fino Grano grueso

MANEJO DE LOS RECURSOS

. La administración de recursos compartidos incluye la asignación de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.

. En sistemas concurrentes es deseable el equilibrio en el acceso a recursos compartidos por todos los procesos (fairness).

. En los programas concurrentes no son deseables:

La **inanición** de un proceso: no logra acceder a los recursos compartidos.

El **overloading** de un proceso: la carga asignada excede su capacidad de procesamiento.

DEADLOCK

. Otro problema importante que se debe evitar es el **deadlock**: por error de programación 2 o mas procesos se quedan esperando que el otro libere un recurso compartido.
. 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 esperan 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.

La no existencia de al menos una de estas 4 condiciones, asegurará la no existencia de deadlock

REQUERIMIENTOS PARA UN LENGUAJE CONCURRENTE

. Los lenguajes de programación concurrente deberán proveer:

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

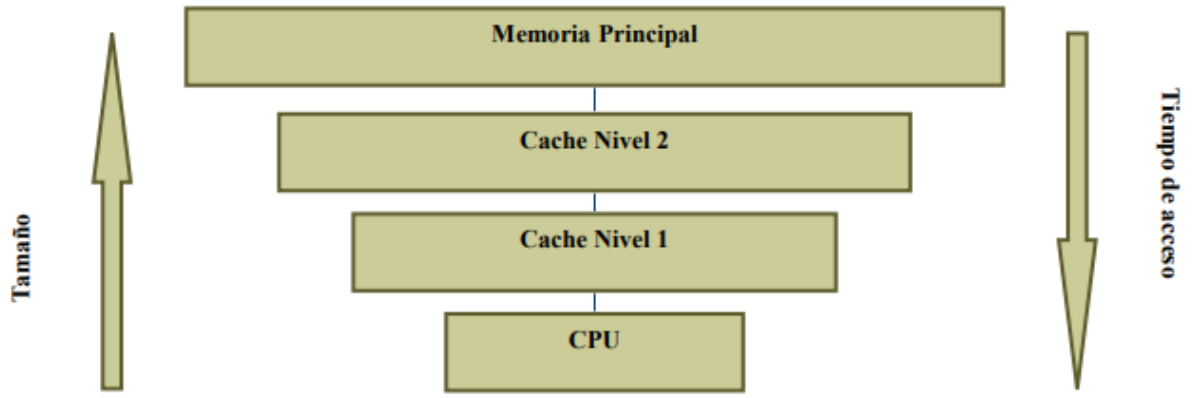
PROBLEMAS ASOCIADOS CON LA PROGRAMACIÓN CONCURRENTE

. 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, lo que supone una 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.

CONCURRENCIA A NIVEL HARDWARE

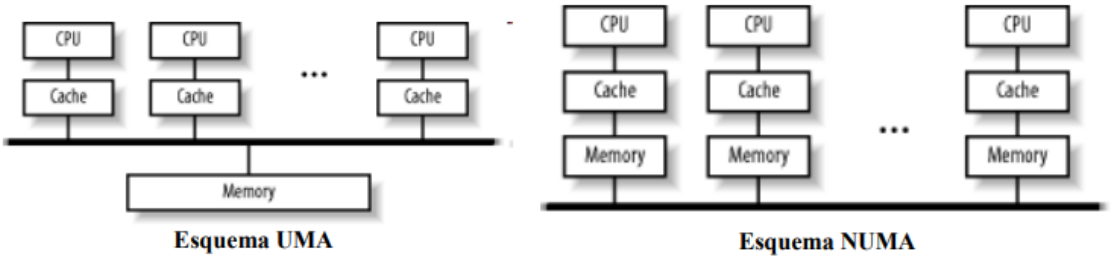
. **Hay un límite físico en la velocidad de los procesadores:** Las máquinas monoprocesador ya no pueden mejorar, entonces, más procesadores por chip para mayor potencia de cómputo. Luego, implementación de Multicores → Cluster de multicores → Consumo => Uso eficiente → Programación concurrente y paralela.
. **Niveles de memoria:** Organizados en jerarquía de memoria. Aparece problema de consistencia por diferencias de tamaño y tiempos de acceso, localidad temporal y espacial de los datos.

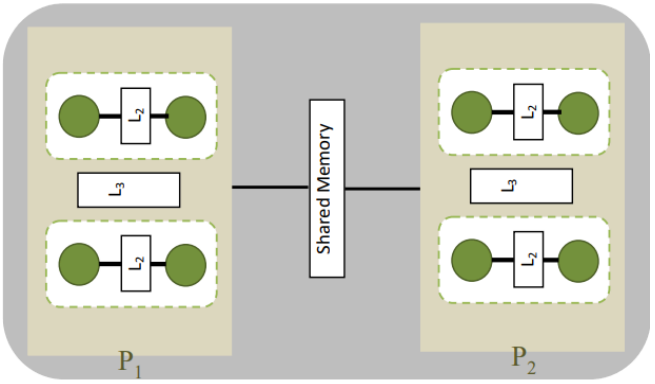
Aparición de máquinas de *Memoria Compartida vs. Memoria distribuida*.



. **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 física 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).

Clase 2 | *ACCIONES ATOMICAS Y SINCRONIZACIÓN*

ATOMICIDAD DE GRANO FINO

. El *Estado* de un programa concurrente es el valor que tienen todas la variables, el valor de los registros, y demás, en un instante de tiempo.
. Cada proceso ejecuta un conjunto de sentencias, cada una de ellas implementada por una o más acciones atómicas. Estas *Acciones Atómicas* hacen transformaciones de estado indivisibles. De esta manera no va a haber interferencias en lo que está haciendo ese proceso por parte de otros. Se produce entonces un *Intercalado (Interleaving)* de acciones atómicas indivisibles.
. Cuando se ejecuta un programa concurrente, la serie de estados por los que se va pasando, forman una *Historia*. Hay muchas posibles historias, pero no todas son válidas.
. La *Interacción* entre los procesos nos va a determinar cuáles historias son correctas.
. La *sincronización por condición* permite restringir las historias de un programa concurrente para asegurar el orden temporal necesario.
. Una acción atómica de grano fino se debe implementar por hardware. Ej: Load PosMemB, reg.
. En la mayoría de los sistemas el tiempo absoluto no es importante, ya que puede haber distintos órdenes (interleavings) en que se ejecutan las instrucciones de los diferentes procesos y los programas deben ser correctos para todos ellos.

ESPECIFICACIÓN DE LA SINCRONIZACIÓN

. *Referencia crítica* en una expresión es una referencia a una variable que es modificada por otro proceso.
. Una expresión *e* que no está en una sentencia de asignación satisface la propiedad de “A lo sumo una vez” si no contiene más de una referencia crítica. *Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez.* De esta manera la ejecución parece atómica.
. Si una expresión o asignación no satisface ASV con frecuencia es necesario un mecanismo de sincronización para construir una acción atómica de grano grueso como una secuencia de acciones atómicas de grano fino que aparecen como indivisibles.
. <e> indica que la expresión e debe ser evaluada atómicamente.
. <await (B) S> se utiliza para especificar sincronización. La expresión booleana B especifica una condición de demora. Una vez que se garantiza que B es *true* comienza la ejecución de S. Ningún estado interno de S es visible para los otros procesos.

. Await general: await (s>0) s=s-1.
. 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);

PROPIEDADES DE SEGURIDAD Y VIDA

. Una propiedad de un programa concurrente es un atributo verdadero en cualquiera de las historias de ejecución del mismo.
. Propiedad de Seguridad (safety): Propone que nada malo le ocurre a un proceso asegurando estados consistentes. Ejemplos de propiedades de seguridad son: exclusión mutua, ausencia de interferencia entre procesos, partial correctness.
. Propiedad de Vida (liveness): Eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks. Ejemplos de vida son: 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

. *Fairness* es el equilibrio en el acceso a los recursos compartidos por todos los procesos, y 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, y una política de scheduling determina cuál será la próxima en ejecutarse.
. **Fairness Incondicional:** Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.
. **Fairness Débil:** Una política de scheduling es débilmente fair si es incondicionalmente fair y si 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.
. **Fairness Fuerte:** Una política de scheduling es fuertemente fair si es incondicionalmente fair y si toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

Clase 3 | *SINCRONIZACIÓN POR MEMORIA COMPARTIDA - LOCKS Y BARRERAS*

. Problemas a resolver:

Problema de la Sección Crítica: implementación de acciones atómicas en software (locks).
Barrera: punto de sincronización que todos los procesos deben alcanzar para que cualquier proceso pueda continuar.
La técnica para resolver estos problemas es la de **Busy Waiting**, donde un proceso chequea repetidamente una condición hasta que sea verdadera.

SECCIÓN CRÍTICA

. Debemos proponer un protocolo de entrada y uno de salida.

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

. Estos protocolos deben satisfacer algunas propiedades:

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 por 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.

. *Solución de "grano fino" Spin Locks:*

Su objetivo es hacer "atómico" el await de grano grueso.
La idea es usar instrucciones como Test & Set (TS), Fetch & Add (FA) o Compare & Swap, disponibles en la mayoría de los procesadores.
Funcionamiento del 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.

. Solución Fair algoritmo Tie-Breaker:

El algoritmo Tie-Breaker 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.

Solución Grano Grueso

```
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 Grano Fino

```
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;
   }
}
```

Ineficiente con N procesos ya que el protocolo de entrada en cada uno es un loop que itera a través de n-1 etapas.

. Solución Fair algoritmo Ticket:

En el 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.

Solución Grano Grueso

Solución Grano Fino

```
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;
   }
}
```

```
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:

En el 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.

Solución Grano Grueso

```
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
      }
}
```

Solución Grano Fino

```
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

. 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.

. Contador Compartido:

n procesos necesitan encontrarse en una barrera. Cada proceso incrementa una variable Cantidad al llegar, y 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;
           FA (cantidad, 1);
           while (cantidad <> n) skip;
         }
}
```

El problema está en encontrar el momento para reiniciar 'cantidad' a 0, en el caso que haya más de una iteración.

. 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). Y se reintroduce contención de memoria, lo que es ineficiente.
Puede usarse un conjunto de valores adicionales y un proceso más, donde cada Worker espera por un único valor.

Solución Grano Grueso

```
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;
             ⟨ await (continuar[i] == 1); ⟩
             continuar[i] = 0;
         }
}

process Coordinador
{ while (true)
        {    for [i = 1 to n]
                  { ⟨ await (arribo[i] == 1); ⟩
                    arribo[i] = 0;
                  }
             for [i = 1 to n] continuar[i] = 1;
         }
}
```

Solución Grano Fino

```
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).

. *Barreras Simétricas:*

Una Barrera Simétrica para n procesos se construye a partir de pares de barreras simples para dos procesos:

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

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

Si n es potencia de 2 se puede implementar *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.

DEFECTOS DE SINCRONIZACIÓN POR BUSY WAITING

. Los protocolos de “busy-waiting” son 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.
. Es muy ineficiente en cuanto a tiempo.
. *Conclusión: ‘Necesitamos herramientas más potentes para diseñar protocolos de sincronización y explotar la concurrencia’.*

Clase 4 | Semáforos

SEMÁFOROS

. Son instancia de un tipo de dato abstracto (o un objeto) con sólo 2 operaciones (métodos) atómicas: P y V.
. Internamente el valor de un semáforo es un entero no negativo:
• V → Señala la ocurrencia de un evento (incrementa).

• P → Se usa para demorar un proceso hasta que ocurra un evento (decrementa).
. Permiten proteger Secciones Críticas y pueden usarse para implementar Sincronización por Condición.

. Exclusión mutua:

```
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;
   }
}
```

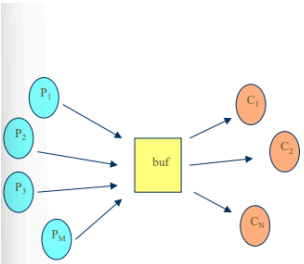
. Barreras - Señalización de eventos:

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

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

process Worker2
{  .........
   V(llega2); P(llega1);
   .........
}
```

. Productores y Consumidores - semáforos binarios divididos:

. Se utiliza la técnica de Semáforo Binario Dividido, donde los semáforos binarios forman un SBS (Split Binary Semaphore) si se cumple el invariante: 0 <= [Suma todos los semáforos] <= 1.
. Las sentencias entre el P y el V ejecutan con exclusión mutua. En general la ejecución de los procesos inicia con un P sobre un semáforo y termina con un V sobre otro de ellos. Esto permite la alternancia entre procesos al acceso a la sección crítica.
. 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:

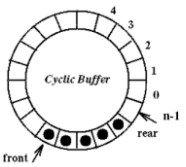
. Cada semáforo cuenta el número de unidades libres de un recurso determinado. Esta forma de utilización es adecuada cuando los procesos compiten por recursos de múltiples unidades.

. 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
       …
    }
}
```

. Depositar y retirar se pudieron asumir atómicas pues sólo hay un productor y un consumidor. El problema aparece cuando hay más de un productor y/o consumidor, ya que podrían estar operando sobre el mismo dato. Podríamos agregar atomicidad para las acciones de cada uno, pero podría seguir habiendo problema entre procesos distintos (consumidores y productores), ya que uno de cada uno podría estar operando sobre el mismo dato. Por esto agregamos exclusión mutua sobre cada slot para evitar retirar dos veces el mismo dato o perderse datos al sobreescribirlo:

```
typeT buf[n];  int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;
sem mutexD = 1, mutexR = 1;
process Productor [i = 1..M]
{ while(true)
    { producir mensaje datos
      P(vacio);
      P(mutexD); buf[libre] = datos; libre = (libre+1) mod n; V(mutexD);
      V(lleno);
    }
}
process Consumidor [i = 1..N]
{ while(true)
    { P(lleno);
      P(mutexR); resultado = buf[ocupado];  ocupado = (ocupado+1) mod n; V(mutexR);
      V(vacio);
      consumir mensaje resultado
    }
}
```

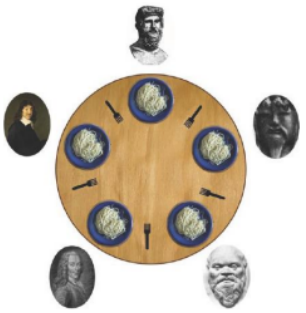


. Varios procesos compitiendo por varios recursos compartidos:

. Tenemos 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, pero puede caerse en deadlock cuando varios procesos compiten por conjuntos superpuestos de recursos.

Problema de los filósofos:

```
process Filosofo [i = 0 to 4]
{  while (true)
      {  adquiere tenedores;
         come;
         libera tenedores;
         piensa;
      }
}
```



. Cada tenedor es una SC, por lo que puede ser tomado por un único filósofo a la vez.
Levantar un tenedor: P. Bajar un tenedor: V.
Cada filósofo necesita el tenedor izquierdo y el derecho.
Si todos compiten por el mismo, pueden bloquearse. Esto lo resolvemos con exclusión mutua selectiva para romper esa cadena.
En este caso con que al menos uno solo compita primero por el tenedor izquierdo mientras que el resto lo hace por el derecho, se rompería la cadena:

```
sem  tenedores [5] = {1,1,1,1,1};

process Filososfos[i = 0..3]
{ while(true)
    {  P(tenedor[i]);  P(tenedor[i+1]);
       comer;
       V(tenedor[i]);  V(tenedor[i+1]);
    }
}

process Filososfos[4]
{ while(true)
    {  P(tenedor[0]);  P(tenedor[4]);
       comer;
       V(tenedor[0]);  V(tenedor[4]);
    }
}
```

. Lectores y escritores - como problema de exclusión mutua:

. Problema: dos clases de procesos (lectores y escritores) comparten una Base de Datos. El acceso de los escritores debe ser exclusivo para evitar interferencia entre transacciones. Los lectores pueden ejecutar concurrentemente entre ellos si no hay escritores actualizando. Es un problema de exclusión mutua selectiva ya que hay distintas clases de procesos que compiten por el acceso a la BD.
. Los escritores necesitan acceso mutuamente exclusivo.
. Los lectores (como grupo) necesitan acceso exclusivo con respecto a cualquier escritor. Sólo el primer lector necesita tomar el lock ejecutando P(rw). Mientras que sera el último lector quien haga V(rw).
. Solución grano grueso:

```
int nr = 0;        # número de lectores activos
sem rw = 1;        # bloquea el acceso a la BD
```

```
process Escritor [j = 1 to N]
{  while(true)
        { …
          P(rw);
          escribe la BD;
          V(rw);
        }
}
```

```
process Lector [i = 1 to M]
{ while(true)
      { …
        < nr = nr + 1;  if (nr == 1) P(rw); >
        lee la BD;
        < nr = nr - 1; if (nr == 0) V(rw);  >
      }
}
```

. Solución grano fino:

```
int nr = 0;        # número de lectores activos
sem rw = 1;        # bloquea el acceso a la BD
sem mutexR= 1;  # bloquea el acceso de los lectores a nr
```

```
process Escritor [j = 1 to N]
{  while(true)
        { …
          P(rw);
          escribe la BD;
          V(rw);
        }
}
```

```
process Lector [i = 1 to M]
{ while(true)
      { …
        P(mutexR);
        nr = nr + 1;
        if (nr == 1) P(rw);
        V(mutexR);
        lee la BD;
        P(mutexR);
        nr = nr - 1;
        if (nr == 0) V(rw);
        V(mutexR);
      }
}
```

. Passing the Baton:

Cuando un proceso está dentro de una SC mantiene el baton (testimonio, token) que significa que tiene permiso para ejecutar. Cuando el proceso llega a un SIGNAL (sale de la SC), pasa el baton 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.
Utiliza SBS.
e es un semáforo binario inicialmente 1 que 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 (dormidos) 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 ⟩

. Lectores y escritores - Técnica Passing the Baton:

Grano Grueso

```
int nr = 0, nw = 0;

process Lector [i = 1 to M]
{ while(true)
    { …
      ⟨ await (nw == 0) nr = nr + 1; ⟩
      lee la BD;
      ⟨ nr = nr – 1; ⟩
    }
}


process Escritor [j = 1 to N]
{ while(true)
    { …
      ⟨ await (nr==0 and nw==0) nw=nw+1; ⟩
      escribe la BD;
      ⟨ nw = nw – 1; ⟩
    }
}
```

Grano Fino

```
int nr = 0, nw = 0, dr = 0, dw = 0;
sem e = 1, r = 0, w = 0;
process Lector [i = 1 to M]
{ while(true) {
        P(e);
        if (nw > 0){dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        SIGNAL1 ;
        lee la BD;
        P(e); nr = nr – 1; SIGNAL2 ;
    }
}
process Escritor [j = 1 to N]
{ while(true) {
        P(e);
        if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
        nw = nw + 1;
        SIGNAL3 ;
        escribe la BD;
        P(e); nw = nw – 1; SIGNAL4 ;
    }
}
```

. El rol de los SIGNALi es el de señalizar exactamente a uno de los semáforos. Algunos de los SIGNAL se pueden modificar.

SIGNAL$_i$ es una abreviación de:

```
if (nw == 0 and dr > 0)
      {dr = dr - 1; V(r);}
elsif (nr == 0 and nw == 0 and dw > 0)
      {dw = dw - 1; V(w);}
else  V(e);
```

. Solución final:

```
int nr = 0, nw = 0, dr = 0, dw = 0;          sem e = 1, r = 0, w = 0:
```

```
process Lector [i = 1 to M]
{  while(true)
     {  P(e);
       if (nw > 0) {dr = dr+1; V(e); P(r); }
       nr = nr + 1;
       if (dr > 0) {dr = dr – 1; V(r); }
       else V(e);
       lee la BD;
       P(e);
       nr = nr – 1;
       if (nr == 0 and dw > 0)
             {dw = dw – 1; V(w); }
       else V(e);
     }
}
```

```
process Escritor [j = 1 to N]
{ while(true)
  {  P(e);
     if (nr > 0 or  nw > 0)
             {dw=dw+1; V(e); P(w);}
     nw = nw + 1;
     V(e);
     escribe la BD;
     P(e);
     nw = nw – 1;
     if (dr > 0) {dr = dr - 1; V(r); }
     elseif (dw > 0) {dw = dw - 1; V(w); }
     else V(e);
  }
}
```

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):*

. 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.
. *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.

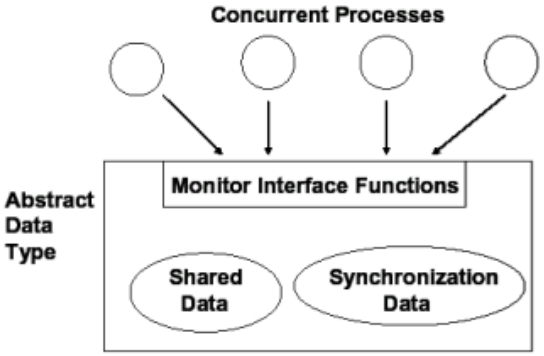
```
bool libre = true;  Pares = set of (int, int) =  ∅;  sem e = 1, b[n] = ([n] 0);

Process Cliente [id: 1..n]
{  int sig;
   //Trabaja
   tiempo = //determina el tiempo de uso del recurso//
   P(e);
   if (! libre) { insertar (tiempo, id) en Pares;
                  V(e);
                  P(b[id]);
                }
   libre = false;
   V(e);
   //USA EL RECURSO
   P(e);
   libre = true;
   if (Pares ≠ ∅ ) { remover el primer par (tiempo, sig) de Pares;
                     V(b[sig]);
                   }
   else V(e);
}
```

¿Que modificaciones deberían
el orden de lle
¿Que modificaciones debe
generalizar la solución a recur

Clase 5 | Monitores

MONITORES

. Son módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.
Estos módulos ofrecen una mayor abstracción de datos ya que encapsulan las representaciones de recursos y brindan un conjunto de operaciones que son los únicos medios para manipular esos recursos.
. *Exclusión Mutua*: se implementa de manera implícita asegurando que los procedures en el mismo monitor no se ejecutan concurrentemente.
. *Sincronización por Condición*: se implementa de manera explícita con variables condición.
. Entonces nuestros programas concurrentes pasan a estar compuestos por procesos activos y monitores pasivos, donde dos procesos interactúan invocando procedures de un monitor.



. Un monitor se diferencia de un TAD en procesos secuenciales, en que el monitor es compartido por procesos que ejecutan concurrentemente.
. Un monitor tiene interfaz y cuerpo:

- La interfaz especifica las operaciones que brinda el recurso.
- El cuerpo tiene variables que representan el estado del recurso y procedures que implementan las operaciones de la interfaz. Estos procedures pueden acceder sólo a variables permanentes, sus variables locales, y parámetros que le sean pasados en la invocación.

. Sincronización:
La sincronización por condición es programada explícitamente 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 re-adquiera el acceso exclusivo al monitor.
- *signal_all(cv)* → despierta todos los procesos demorados en cv, quedando vacía la cola asociada a cv.

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.

EJEMPLOS Y TÉCNICAS CON MONITORES

. Simulación de semáforos - condición básica:

```
monitor Semaforo
{  int s = 1;   cond pos;

   procedure  P ()
      { while (s == 0) wait(pos);
        s = s-1;
      };

   procedure V ()
      { s = s+1;
        signal(pos);
      };
};
```

La diferencia con los semáforos reales es que cuando se realiza el SIGNAL o V(), la competencia se reduce a solo aquellos procesos que se encuentran dentro del monitor.
Para hacer una simulación idéntica, deberíamos hacer un SIGNAL_ALL en el procedimiento V().

. Simulación de semáforos - Passing the Conditions:

```
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);}
     };
};
```

Respeta el orden de llegada de los procesos que se encuentran dormidos.
Al hacer un V(), si no hay nadie esperando (dormido), incrementa el valor de *s* en 1. Pero si hay alguien dormido, solo decrementa el contador de procesos en espera y despierta al ultimo que llego.

. Alocación SJN - 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]);
              };
     };
}
```

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

. Buffer Limitado - Sincronización por Condición Básica:

```
monitor Buffer_Limitado
{  typeT buf[n];
   int ocupado = 0, libre = 0;  cantidad = 0;
   cond not_lleno, not_vacio;

   procedure depositar(typeT datos)
      { while (cantidad == n) wait (not_lleno);
        buf[libre] = datos;
        libre = (libre+1) mod n;
        cantidad++;
        signal(not_vacio);
      }

   procedure retirar(typeT &resultado)
      { while (cantidad == 0) wait(not_vacio);
        resultado=buf[ocupado];
        ocupado=(ocupado+1) mod n;
        cantidad--;
        signal(not_lleno);
      }
}
```

Para un buffer de N posiciones, a la hora de depositar, si la var *cantidad* es igual a N, el proceso deberá dormirse en una cola condicion de procesos listos para depositar.
Por otro lado, a la hora de retirar, si la var *cantidad* es igual a 0, el proceso deberá dormirse en una cola de procesos listos para retirar.
Luego, el proceso que deposita, depositara en la posición indicada por la var *libre* e incrementara la misma de manera circular para que indique a la próxima posición libre.
Posteriormente incrementa *cantidad* y despierta a uno de los procesos que están listos para retirar.
Los procesos que retiren deberán tomar el dato de la posición indicada por la var *ocupado* y luego incrementarla de igual manera que *libre*. Después decrementar cantidad y despertaran a uno de los procesos que estén listos para depositar.

. Lectores y escritores:

. Solución Broadcast Signal:

```
monitor Controlador_RW
{  int nr = 0, nw = 0;
   cond  ok_leer, ok_escribir
   procedure pedido_leer( )
      { while (nw > 0) wait (ok_leer);
        nr = nr + 1;
      }
   procedure libera_leer( )
      { nr = nr - 1;
        if (nr == 0) signal (ok_escribir);
      }
   procedure pedido_escribir( )
      { while (nr>0 OR nw>0) wait (ok_escribir);
        nw = nw + 1;
      }
   procedure libera_escribir( )
      { nw = nw - 1;
        signal (ok_escribir);
        signal_all (ok_leer);
      }
}
```

El monitor arbitra el acceso a la BD.

Los procesos dicen cuándo quieren acceder y cuándo terminaron. Y requieren un monitor con 4 procedures: pedido_leer - libera_leer - pedido_escribir - libera_escribir.

. Solución 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);
                   }
            }
     }
}
```

Son los procesos los encargados de despertar a quienes están esperando.

. Diseño de un reloj lógico - 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);
             }
      }
}
```

Se utilizan 2 colas condición, una general y una particular a cada proceso.
Los procesos entran y salen de las colas constantemente.

. Peluquero dormilón - Rendezvous:

```
monitor Peluqueria {
    int peluquero = 0, silla = 0, abierto = 0;
    cond peluquero_disponible, silla_ocupada, puerta_abierta, salio_cliente;
    procedure corte_de_pelo() {
        while (peluquero == 0)   wait (peluquero_disponible);
        peluquero = peluquero - 1;
        signal (silla_ocupada);
        wait (puerta_abierta);
        signal (salio_cliente);
    }
    procedure proximo_cliente(){
         peluquero = peluquero + 1;
         signal(peluquero_disponible);
         wait(silla_ocupada);
    }
    procedure corte_terminado() {
         signal(puerta_abierta);
         wait(salio_cliente);
    }
}
```

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

va. Si hay clientes esperando, el peluquero despierta a uno y espera que se siente. Sino, se vuelve a dormir hasta que llegue un cliente.
El peluquero y el cliente atraviesan una serie de etapas de sincronización, comenzando con un rendezvous similar a una barrera entre dos procesos, pues ambas partes deben arribar antes de que cualquiera pueda seguir.

. Scheduling de Disco:
. Solucion Monitor Separado:
El scheduler es implementado por un monitor para que los datos sean accedidos solo por un proceso usuario a la vez. El monitor provee dos operaciones: pedir y liberar.
Un proceso usuario que quiere acceder al cilindro cil llama a pedir(cil), y retoma el control cuando el scheduler seleccionó su pedido. Luego, el proceso usuario accede al disco (llamando a un procedure o comunicándose con un proceso manejador del disco). Luego de acceder al disco, el usuario llama a liberar: Scheduler_Disco.pedir(cil) - Accede al disco - Scheduler_Disco.liberar( ) .

```
monitor Scheduler_Disco
{ int posicion = -1, v_actual = 0, v_proxima = 1;
  cond scan[2];

  procedure pedir(int cil)
    { if (posicion == -1)  posicion = cil;
      elseif (cil > posicion)   wait(scan[v_actual],cil);
      else  wait(scan[v_proxima],cil);
    }

  procedure liberar()
      { if (!empty(scan[v_actual]))  posicion = minrank(scan[v_actual]);
        elseif (!empty(scan[v_proxima]))
            { v_actual :=: v_proxima;
              posicion = minrank(scan[v_actual]);
            }
        else  posicion = -1;
        signal(scan[v_actual]);
      }
}
```

Problema: La presencia del scheduler es visible al proceso que usa el disco. Si se borra el scheduler, los procesos usuario cambian. Además, todos los procesos usuario deben seguir el protocolo de acceso. Si alguno no lo hace, el scheduling falla. Y por último, luego de obtener el acceso, el proceso debe comunicarse con el driver de acceso al disco a través de 2 instancias de buffer limitado.

. Solucion Monitor Intermedio:
Utiliza un monitor como intermediario entre los procesos usuario y el disk driver. El monitor envía los pedidos al disk driver en el orden de preferencia deseado.
Mejoras: La interfaz al disco usa un único monitor, y los usuarios hacen un solo llamado al monitor por acceso al disco. La existencia o no de scheduling es transparente. Y además, no hay un protocolo multipaso que deba seguir el usuario y en el cual pueda fallar.

```
monitor Interfaz_al_disco
{  int posicion = -2, v_actual = 0, v_proxima = 1, args = 0, resultados = 0;
   cond scan[2];
   cond args_almacenados, resultados_almacenados, resultados_recuperados;
   argType area_arg; resultadoType area_resultado;

   procedure usar_disco (int cil; argType params_transferencia;resultType &params_resultado)
      {  if (posicion == -1)    posicion = cil;
         elseif (cil > posicion)   wait(scan[v_actual],cil);
         else wait(scan[v_proxima],cil);
         area_arg = parametros_transferencia;
         args = args+1; signal(args_almacenados);
         wait(resultados_almacenados);
         parametros_resultado = area_resultado;
         resultados = resultados-1;
         signal(resultados_recuperados);
      }

procedure buscar_proximo_pedido (argType &parametros_transferencia)
      { int temp;
         if (!empty(scan[v_actual]))  posicion = minrank(scan[v_actual]);
         elseif (!empty(scan[v_proxima]))
                { v_actual :=: v_proxima;
                  posicion = minrank(scan[v_actual]);
                }
         else  posicion = -1;
         signal(scan[v_actual]);
         if (args == 0) wait(args_almacenados);
         parametros_transferencia = area_arg; args = args-1;
       }

   procedure transferencia_terminada (resultType valores_resultado)
      { area_resultado := valores_resultado;
         resultados = resultados+1;
         signal(resultados_almacenados);
         wait(resultados_recuperados);
      }
}
```

EXCLUSIÓN MUTUA SELECTIVA

En los problemas de exclusión mutua selectiva, cada proceso compite por sus recursos no con todos los demás procesos sino con un subconjunto de ellos.
Dos casos típicos de dicha competencia se producen cuando los procesos compiten por los recursos según su tipo de proceso o por su proximidad. Por ejemplo los problemas de lectores/escritores y filósofos.
En el problema de lectores/escritores se da exclusión mutua entre clases de procesos: los lectores como clase de proceso compiten con los escritores, ya que cuando un lector está accediendo a la BD los demás lectores también pueden hacerlo, pero se excluye a los escritores.

Clase 6 | Programación Concurrente en Memoria Distribuida

GENERALIDADES

. Las *Arquitecturas de memoria distribuida* son el resultado de: procesadores + memoria local + red
de comunicaciones + mecanismo de comunicación / sincronización, manejado a través del intercambio de mensajes.
. *Programa distribuido*: es un programa concurrente comunicado por mensajes. Supone la ejecución sobre una arquitectura de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida).
. Primitivas de *pasaje de mensajes*: es necesaria una interfaz con el sistema de comunicaciones que provea de { semáforos + datos + sincronización }.
. Los procesos SOLO comparten *canales* (físicos o lógicos). Variantes para los canales:

- Tipos de a cuerdo a quienes pueden enviar mensajes y quienes pueden recibir:
  Mailbox: canales totalmente compartidos donde todos los procesos pueden enviar y recibir mensajes.
  Input port: hay un único proceso receptor, pero cualquiera puede enviar información.
  Link (punto a punto): relaciones 1 a 1, donde un único proceso puede enviar información y un único proceso puede recibir.
- Clasificación según los sentidos en que viaja la información:
  Unidireccionales: la información viaja en un solo sentido, pudiendo sólo enviar información. Si necesitamos enviar y recibir información, deberíamos tener 2 canales por separado.
  Bidireccionales: la información viaja en ambos sentidos.
- Clasificación de acuerdo a la manera de comunicarse:
  Sincrónicos: los procesos deben establecer la comunicación en un punto de encuentro, y ninguno puede continuar hasta que la comunicación haya finalizado.
  Asincrónicos: la comunicación no requiere de ambos procesos trabajando a la vez, sino que el proceso que envía el mensaje podría dejarlo y continuar su procesamiento sin tener que esperar a que llegue el receptor.

. Los canales, entonces, son lo único que comparten los procesos, donde las variables son locales a un proceso (“proceso cuidador”), y donde la exclusión mutua no requiere mecanismo especial (está implícita).
. Los Mecanismos para el Procesamiento Distribuido son:

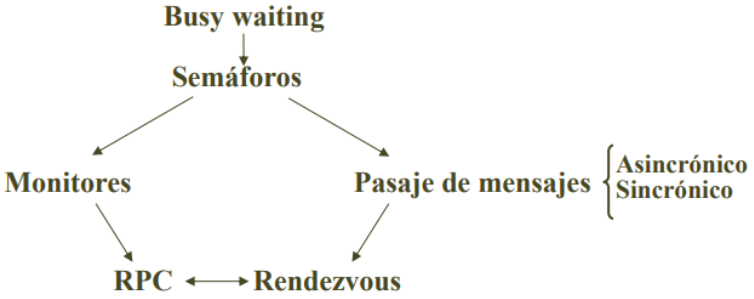
- Pasaje de Mensajes Asincrónicos (PMA)
- Pasaje de Mensajes Sincrónico (PMS)
- Llamado a Procedimientos Remotos (RPC)
- Rendezvous

. La sincronización de la comunicación interproceso depende del patrón de interacción:

- Productores y consumidores (Filtros o pipes)
- Clientes y servidores
- Pares que interactúan

. 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 ASINCRONICOS (PMA)

. Uso de canales 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 Receive* → un proceso recibe un mensaje desde un canal con receive, que demora (“bloquea”) al receptor hasta que en el canal haya al menos un mensaje; luego toma el primero y lo almacena en variables locales: receive ch(var1 , ... , varn);
Las variables del receive deben tener los mismos tipos que la declaración del canal.
Receive es una primitiva bloqueante.

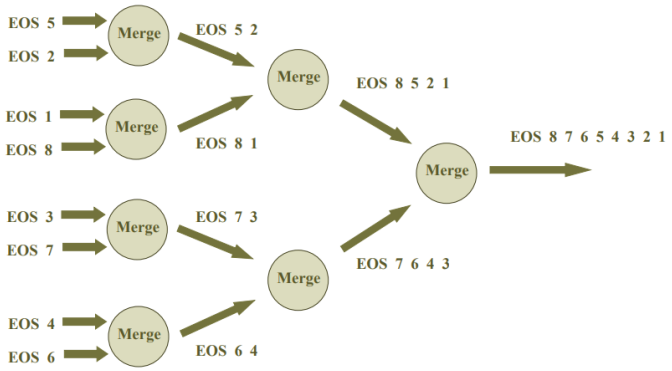
. Características de canales PMA:
Acceso a los contenidos de cada canal: atómico y respeta orden FIFO.
La función *empty(ch)* → determina si la cola de un canal está vacía. Es útil cuando el proceso puede hacer trabajo productivo mientras espera un mensaje. Pero debe usarse con cuidado, ya que:
- Podría ser false, pero a la vez que el proceso ejecuta el *receive()* otro también lo hace y le quita el mensaje, dejando el canal más mensajes cuando el primer proceso sigue ejecutando (si no es el único en recibir por ese canal).
- O bien, la evaluación de empty podría ser true, y sin embargo no existir un mensaje al momento de que el proceso reanuda la ejecución.

Los canales son declarados globales a los procesos, ya que pueden ser compartidos.

. Ejemplo Productores y Consumidores:
Problema: ordenar una lista de N números de modo ascendente. Podemos pensar en un filtro Sort con un canal de entrada (N números desordenados) y un canal de salida (N números ordenados).
Este **Filtro** es un proceso que recibe mensajes de uno o más canales de entrada y envía mensajes a uno o más canales de salida. La salida de un filtro es función de su estado inicial y de los valores recibidos.
Una solución más eficiente que la “secuencial” podría ser una red de pequeños procesos que ejecutan en paralelo e interactúan para “armar” la salida ordenada (merge network).

Entonces, la idea sería mezclar repetidamente y en paralelo dos listas ordenadas de N1 elementos cada una en una lista ordenada de 2*N1 elementos.
Con PMA, pensamos en 2 canales de entrada por cada canal de salida, y un carácter especial EOS cerrará cada lista parcial ordenada.

```
chan  in1(int), in2(int), out(int);

Process Merge
{ int v1, v2;

  receive in1(v1);
  receive in2(v2);
  while (v1 ≠ EOS) and (v2 ≠ EOS)
    { if  (v1 ≤ v2) { send out(v1); receive in1(v1); }
      else  { send out(v2); receive in2(v2); }
    }

  if  (v1 == EOS) while (v2 ≠ EOS) {send out(v2); receive in2(v2);}
  else  while (v1 ≠ EOS) {send out(v1); receive in1(v1);}

  send out (EOS);
}
```



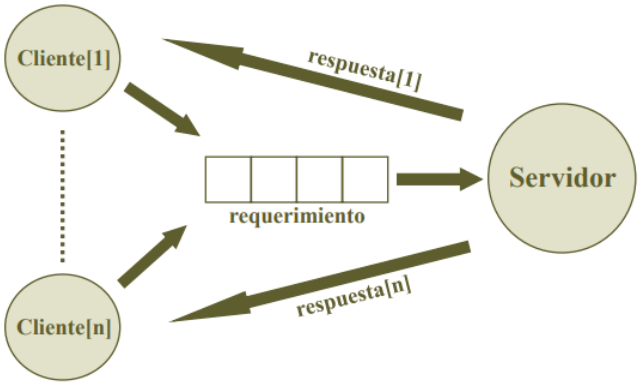
Puede programarse usando:

- Static naming (arreglo global de canales, y cada instancia de Merge recibe desde 2 elementos del arreglo y envía a otro embeber el árbol en un arreglo).
- Dynamic naming (canales globales, parametrizar los procesos, y darle a cada proceso 3 canales al crearlo; todos los Merge son idénticos, pero se necesita un coordinador).

. Ejemplo Cliente/Servidor:
Simulamos un *Monitor*, el cual manejara recursos, encapsulando variables permanentes que registran el estado, y proveyendo de un conjunto de procedures. Para simularlos utilizaremos 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.
Un proceso Cliente que envía un mensaje a un canal de requerimientos general, luego recibe el resultado desde un canal de respuesta propio.
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).

_ Monitores Activos – 1 operación:

- Para simular *Monitor*, 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.




```
chan requerimiento (int idCliente, tipos de los valores de entrada );
chan respuesta[n] (tipos de los resultados );

Process Servidor
      {  int idCliente;
         declaración de variables permanentes;
         código de inicialización;
         while (true)
            { receive requerimiento (IdCliente, valores de entrada);
              cuerpo de la operación op;
              send respuesta[IdCliente] (resultados);
            }
      }

Process Cliente [i = 1 to n]
      {  send requerimiento (i, argumentos);
         receive respuesta[i] (resultados);
      }
```

_ *Monitores Activos – 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*.

```
type clase_op = enum(op1, ..., opn);
type tipo_arg = union(arg1 : tipoAr1, ..., argn : tipoArn );
type tipo_result = union(res1 : tipoRe1, ..., resn : tipoRen );

chan request(int idCliente, clase_op, tipo_arg);
chan respuesta[n](tipo_result);

Process Servidor .....
Process Cliente [i = 1 to n]  .....
```

```
Process Servidor
{  int IdCliente; clase_op oper; tipo_arg args;
   tipo_result resultados;
   código de inicialización;
   while ( true)
      {  receive request(IdCliente, oper, args);
         if ( oper == op1 ) { cuerpo de op1;}
            .....
         elsif ( oper == opn ) { cuerpo de opn;}
         send respuesta[IdCliente](resultados);
      }
}
Process Cliente [i = 1 to n]
{  tipo_arg mis_args;
   tipo_result  mis_resultados;
   send request(i, opk, mis_args);
   receive respuesta[i] (mis_resultados);
}
```

_ *Monitores Activos – Múltiples operaciones y variables condición:*

Consideramos un caso específico de manejo de múltiples unidades de un recurso (ejemplos: bloques de memoria, impresoras). Donde:

- 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.

```
type clase_op = enum(adquirir, liberar);
chan request(int idCliente, claseOp oper, int idUnidad );
chan respuesta[n] (int id_unidad);

Process Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades = valor inicial disponible;
    queue pendientes;
    while (true)
      {  receive request (IdCliente, oper, id_unidad);
         if (oper == adquirir)
            {   if (disponible > 0)
                  {  disponible = disponible - 1;
                     remove (unidades, id_unidad);
                     send respuesta[IdCliente] (id_unidad);
                  }
               else  push (pendientes, IdCliente);
            }
         else
            { if empty (pendientes)
                 {  disponible= disponible + 1;
                    insert(unidades, id_unidad);
                 }
              else
                 {  pop (pendientes, IdCliente);
                    send respuesta[IdCliente](id_unidad);
                 }
            }
      } //while
}   //process  Administrador_Recurso

Process Cliente[i = 1 to n]
{   int id_unidad;
    send request(i, adquirir, 0);
    receive respuesta[i](id_unidad);
     //Usa la unidad
    send request(i, liberar, id_unidad);
}
```

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( ); receive respuesta*
• Entry del monitor	↔	*receive request( )*
• Retorno del procedure	↔	*send respuesta( )*
• Sentencia *wait*	↔	Salvar pedido pendiente
• Sentencia *signal*	↔	Recuperar/ procesar pedido pendiente
• Cuerpos de los procedure	↔	Sentencias del "case" de acuerdo a la clase de operación.

. Con sentencia de Alternativa Múltiple:

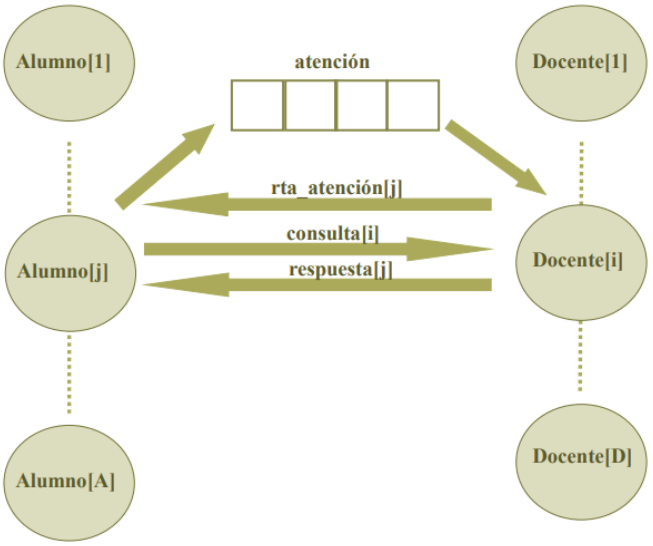
```
chan pedido (int idCliente);
chan liberar (int idUnidad);
chan respuesta[n] (int idUnidad);

Process Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades = valor inicial disponible;
    int id_unidad, idCliente;
    while (true)
     {  if  ( (not empty(pedido) and (disponible >0) ) →
              receive pedido (idCliente);
              disponible = disponible - 1;
              remove (unidades, id_unidad);
              send respuesta[idCliente] (id_unidad);
        □ (not empty(liberar)) →
              receive liberar (id_unidad);
               disponible= disponible + 1;
               insert(unidades, id_unidad);
        } //if
     } //while
}    //process  Administrador_Recurso
```

```
Process Cliente[i = 1 to n]
 {  int id_unidad;

    send pedido (i);
    receive respuesta[i](id_unidad);
     //Usa la unidad
    send liberar (id_unidad);
 }
```

_ *Continuidad Conversacional:*

Problema: 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.

Todos los alumnos pueden pedir atención por un *canal globa*l y recibirán respuesta de un docente dado por un *canal propio*.



Solución:

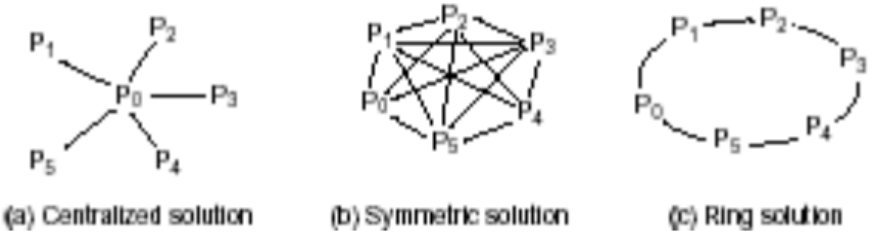
```
chan atención (int);
chan consulta[D] (string);
chan rta_atención[A](int);
chan respuesta[A] (string);

Process Alumno [a = 1 to A]
    {  int idDocente;
       string preg, res;
       send atención (a);
       receive rta_atención[a] (idDocente);
       while (tenga consultas para hacer)
          { send consulta[idDocente](preg);
            receive respuesta[a](res);
          }
       send consulta [idDocente] ('FIN');
    }
```

```
Process Docente [d = 1 to D]
{    string preg, res;
     int idAlumno;
     bool seguir = false;

     while (true)
       {  receive atención (idAlumno);
          send rta_atención[idAlumno](d);
          seguir = true;
          while (seguir)
             { receive consulta[d](preg);
               if (preg == 'FIN')  seguir = false
               else
                    { res = resolver la pregunta (preg)
                      send respuesta [idAlumno](res);
                    }
             }
       }
}
```

. Pares (Peers) Interactuantes:

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.

Tres posibles arquitecturas:



_ *Solución Centralizada:*

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, dejando un saldo de 2(N-1) mensajes.

```
chan valores(int),   resultados[n-1] (int minimo, int maximo);
```

```
Process P[0]
{  int v; int nuevo, minimo = v, máximo = v;
   for [i=1 to n-1]
      {     receive valores (nuevo);
            if (nuevo < minimo)  minimo = nuevo;
            if (nuevo > maximo)  maximo = nuevo;
      }
   for [i=1 to n-1]
        send resultados [i-1] (minimo, maximo);
}
```

```
Process P[i=1 to n-1]
{  int v;  int minimo, máximo;
   send valores (v);
   receive resultados[i-1](minimo, maximo);
}
```

¿Se puede usar un único canal de resultados? Si, pero de todas formas sigue requiriendo que el proceso 0 envíe el resultado N-1 veces.

_ *Solución Simétrica:*

En la arquitectura simétrica o “full connected” hay un canal entre cada par de procesos. Todos los procesos ejecutan el mismo algoritmo.

Cada proceso transmite 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.

```
chan valores[n] (int);
Process P[i=0 to n-1]
{ int v=…, nuevo, minimo = v, máximo=v;
  for [k=0 to n-1  st  k <> i ]
      send valores[k] (v);
  for [k=0 to n-1  st  k <> i ]
     { receive valores[i] (nuevo);
       if (nuevo < minimo) minimo = nuevo;
       if (nuevo > maximo) maximo = nuevo;
     }
}
```

¿Se puede usar un único canal? No, porque si todos envían sus valores a través del mismo canal se mezclarian todos y no se sabría de cada valor de quien es y para quien es.

_ *Solución en Anillo Circular:*

Esta solución consiste en 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, transmitiendo 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, ya que el envío de los mensajes es secuencial, por lo que no hay procesamiento concurrente.

```
chan valores[n] (int minimo, int maximo);
Process P[0]
{ int v=…, minimo = v, máximo=v;
  send valores[1] (minimo, maximo);
  receive valores[0] (minimo, maximo);
  send valores[1] (minimo, maximo);
}

Process P[i=1 to n-1]
{ int v=…, minimo, máximo;
  receive valores[i] (minimo, maximo);
  if (v<minimo) minimo = v;
  if (v> maximo) maximo = v;
  send valores[(i+1) mod n] (minimo, maximo);
  receive valores[i] (minimo, maximo);
  if (i < n-1) send valores[i+1] (minimo, maximo);
 };
```

. Conclusión:

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 7 | PMS, CPS, Paradigmas de Interacción entre Procesos

PASAJE DE MENSAJES SINCRÓNICOS (PMS)

. Ahora los canales son de tipo link o punto a punto. Solo tienen 1 emisor y 1 receptor.
Es decir, que si necesitamos enviar información del proceso A al proceso B y luego recibir información también del proceso B, vamos a necesitar 2 canales, uno para el envío y otro para la recepción.
. La diferencia entre PMA y PMS es la primitiva de transmisión Send. En PMS es bloqueante:

- 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).

. Ejemplo: productor / consumidor:

```
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
      }
}
```

Se utiliza un único canal que tiene a 'Productor' como emisor y a 'Consumidor' como receptor.
El 'Productor' genera un valor, lo manda al 'Consumidor', pero hasta que el 'Consumidor' no lo recibe no puede empezar a producir un nuevo valor.

. PMA vs. PMS:
. Hay mayor concurrencia en PMA, ya que:
- 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.

. Para lograr el mismo efecto en PMS se debe interponer un proceso "buffer".

CPS - LENGUAJE PARA PMS

. Las ideas básicas introducidas por Hoare fueron PMS y comunicación guardada, utilizando PM con waiting selectivo dependiendo el estado en que están los procesos y los mensajes.
. Canal: link directo entre dos procesos en lugar de mailbox global. Son halfduplex (comunicaciones en una única dirección) y nominados, es decir, que no tienen que ser declarados, solo se hace referencia al otro proceso.
. Las sentencias de Entrada (?) y Salida (!) son el único medio por el cual los procesos se comunican. Para que se produzca la comunicación, deben matchear, y luego se ejecutan simultáneamente.
- Sentencia de entrada: Destino ! port(e1 , ..., en);
- Sentencia de salida: Fuente ? port(x1 , ..., xn);

El puerto se puede especificar o no. Si se especifica, también debe *matchear*.

. Comunicación Guardada:
Hay limitaciones de ? y ! ya que son bloqueantes. Hay problemas si un proceso quiere comunicarse con otros (quizás por ports) sin conocer el orden en que los otros quieren hacerlo con él.
Entonces, las operaciones de comunicación (? y !) pueden ser guardadas, es decir hacer un AWAIT hasta que una condición sea verdadera.

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.

. Ejemplo:

Las sentencias de comunicación guardadas aparecen en if y do.

```
if
    B1 ; comunicación 1 → S1 ;
    B2 ; comunicación 2 → S2 ;
fi
```

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 Si.

PARADIGMAS DE INTERACCIÓN ENTRE PROCESOS

. Manager/Worker:

- Implementación distribuida del modelo Bag of Task.
- 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.
- La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los workers.
- Vamos a tener 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.

. Heartbeat:

- Útil para soluciones iterativas que se quieren paralelizar, usando un esquema “divide & conquer” donde se distribuye la carga (datos) entre los workers; cada uno responsable de actualizar una parte. Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos.
- Cada “paso” debería 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;
    }
```

}

• La idea principal de este algoritmo es que se expande enviando información; luego se contrae incorporando nueva información.

. *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 un esquema **abierto** (W1 en el INPUT, Wn en el OUTPUT), y se encuentran en procesadores que operan en paralelo

• Pueden estar, sino, en un esquema **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.

• O pueden estar en un esquema **cerrado**, donde existe un proceso coordinador que maneja la “realimentación” entre Wn y W1 .

. *Prueba-eco:*

• Árboles y grafos son utilizados en muchas aplicaciones distribuidas como búsquedas en la WEB, BD, sistemas expertos y juegos. 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 en un árbol o grafo. Este paradigma es el análogo concurrente de DFS.

• Prueba-eco se basa en el envío de un mensaje (“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 (o 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.

• 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.

. *Token Passing:*

• Este paradigma utiliza 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, ya que solo quien tenga el token podrá acceder a la información en un determinado momento.

• 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.

. *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 8 | RPC y Rendezvous

. El Pasaje de Mensajes se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la comunicación unidireccional.
. Pero, para resolver Cliente/Servidor, la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas). Además, cada cliente necesita un canal de reply distinto.
. RPC (Remote Procedure Call) y Rendezvous son técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional, y que son 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, donde se demora al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados.
. La diferencia entre RPC y Rendezvous es que, RPC declara un procedure para cada operación y luego crea un proceso nuevo para manejar cada llamado. Mientras que Rendezvous es servido por una sentencia de Entrada (o accept) que espera una invocación, la procesa y devuelve los resultados.

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 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 en una declaración proc:

```
proc opname(identif. formales) returns identificador resultado
  declaración de variables locales
  sentencias
end
```

. Un 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 omitir.

. 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, en este caso se debe crear un proceso server o alocarlo de un pool preexistente.

. Sincronización de módulos:

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

Aunque un proceso llamador y su server se sincronizan, el único rol del server es actuar en nombre del llamador (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: puede llevar a problemas de exclusión mutua y demás.

RENDEZVOUS

. Combina comunicación y sincronización:

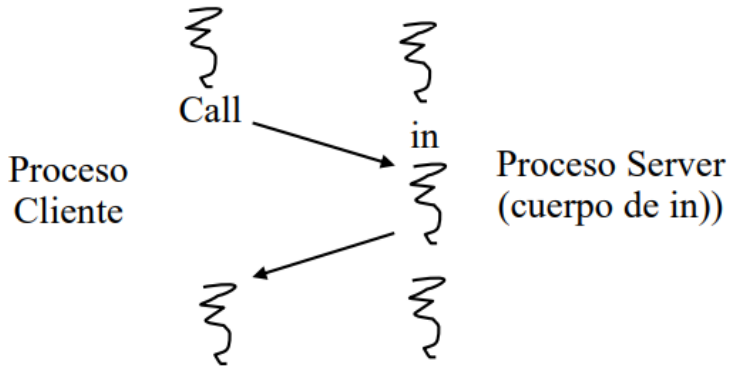
- Como con RPC, un proceso cliente invoca una operación por medio de un call, pero esta operación es servida por un proceso existente en lugar de por uno nuevo.
- Un proceso servidor usa una sentencia de entrada para esperar por un call y actuar.
- Las operaciones se atienden una por vez más que concurrentemente

. 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:



. Se puede combinar comunicación guardada con rendezvous:

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

- Los Bi son expresiones de sincronización opcionales.
- Los ei son expresiones de scheduling opcionales.

ADA - LENGUAJE CON RENDEZVOUS

. 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).

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 el TASK.

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

• Call: Entry Call:
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 con 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 B1 ⇒ accept E1; sentencias1
or     ...
or     when Bn ⇒ accept En; sentencias n
end select;
```

- Cada línea se llama alternativa. Las cláusulas when son opcionales.
- Puede contener una alternativa else, or delay, or terminate.
- Uso de atributos del entry: count, calleable.

Clase 9 | Pthreads - Sem. y Monit. en Pthreads - Librería para PM (MPI)

PTHREADS

. Thread: proceso “liviano” que tiene su propio contador de programa y su pila de ejecución, pero no controla el “contexto pesado”.
. Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones “multithreading”.
Al principio, todos los mecanismos eran diferentes, por lo que era necesaria una unificación, entonces a mediados de los 90, POSIX auspició el desarrolló de una biblioteca en C para multithreading, *Pthreads*. Esta es una biblioteca para programación paralela en memoria compartida, que permite crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

. Funciones básicas:

```
#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.
. Las secciones críticas se implementan en Pthreads utilizando mutex locks (bloqueo por exclusión mutua) por medio de variables mutex.
. Una variable mutex tiene dos estados: locked (bloqueado) y 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.

. Funciones de la API threads 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);
```

. Pthreads soporta tres tipos de Mutexs (Locks):

- Normal: Un Mutex con el atributo Normal NO permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él (deadlock).
- Recursivo: Un Mutex con el atributo Recursive SI permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él. Simplemente incrementa una cuenta de control.
- ErrorCheck: 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.
. 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).
```

. 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 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).

• Funciones para manejar Variables Condición en Pthreads:

```
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.

Funciones para manejar Attribute Objects:

```
int pthread_attr_init (pthread_attr_t  *attr);

int pthread_attr_destroy (pthread_attr_t  *attr);
```

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);
```

Type especifica el tipo de mutex y puede tomar los valores:

```
PTHREAD_MUTEX_NORMAL_NP

PTHREAD_MUTEX_RECURSIVE_NP

PTHREAD_MUTEX_ERRORCHECK_NP
```

SEMÁFOROS EN PTHREADS

. Los threads se 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.

MONITORES EN PTHREADS

. Pthreads no posee “Monitores”, pero con las herramientas que provee se puede simular el uso de monitores: con mutex se hace la exclusión mutua que nos brindaba implícitamente el monitor, y con las variables condición la sincronización:

✓ El acceso exclusivo 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.

LIBRERIA PARA MANEJO DE PM

. Los prototipos de las operaciones son:

*Send (void *sendbuf, int nelems, int dest)*

*Receive (void *recvbuf, int nelems, int source)*

. Protocolos para Send/Receive:

• Send/Receive Bloqueante: No se devuelve el control del Send hasta que el dato a transmitir esté seguro, por lo que el proceso que hace el Send se bloquea. Esto genera ociosidad del proceso emisor.

Hay dos posibilidades:

- Send/Receive bloqueantes sin buffering: la comunicación se asemeja a Pasaje de Mensajes Sincronico. Es decir que los procesos se esperan mutuamente.
- Send/Receive bloqueantes con buffering: hay un buffer donde se van almacenando los mensajes. Ese buffer está en algún lugar de la memoria, pero no hay una memoria compartida. Se necesita, entonces, definir donde va a estar el buffer. Por lo general se encuentra del lado del receptor.

• Send/Receive No Bloqueante: Para evitar overhead (ociosidad o manejo de buffer) se devuelve el control de la operación inmediatamente. Por lo que requiere un posterior chequeo para asegurarse la finalización de la comunicación. Este chequeo queda en manos del programador, para asegurar la semántica del SEND.

Hay dos posibilidades:

- Send/Receive no bloqueantes sin buffering: hasta que el otro proceso no hizo el *receive* la comunicación no se hace, cuando el otro proceso llega al *receive*, recién ahí se hace la comunicación real.

  Por lo que desde que el proceso emisor hace el *send* y el proceso receptor hace el *receive*, el dato está inseguro.

- Send/Receive no bloqueantes con buffering: se va a tener un buffer del lado del receptor. El tiempo que el dato va a estar inseguro es el tiempo que pasa entre el *send* y que el dato es guardado en el buffer.
Varía el tiempo que esté seguro o no el dato.

MESSAGE PASSING INTERFACE (LIBRERIA MPI)

. MPI define una librería estándar que puede 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.
. 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)*

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, y existe un comunicador que incluye a todos los procesos de la aplicación MPI_COMM_WORLD.
Son variables del tipo MPI_Comm → almacena información sobre qué procesos pertenecen a él.
En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar.

. Adquisicion de Informacion:
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:*

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 a 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._

Estructura MPI_Status:

typedef struct MPI_Status {int MPI_SOURCE;
int MPI_TAG;
int MPI_ERROR; }

MPI_Get_count para obtener la cantidad de elementos recibidos:

*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_Test (MPI_Request *solicitud, int *flag, MPI_Status *estado)*

MPI_Wait: bloquea al proceso hasta que finaliza la operación.

*MPI_Wait (MPI_Request *solicitud, MPI_Status *estado)*

Este tipo de comunicación permite solapar cómputo 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_Probe (int origen, int tag, MPI_Comm comunicador, MPI_Status *estado)*

MPI_Iprobe: chequea por el arribo de un mensaje que cumpla con el origen y tag.

*MPI_Iprobe (int origen, int tag, MPI_Comm comunicador, int *flag, MPI_Status *estado)*

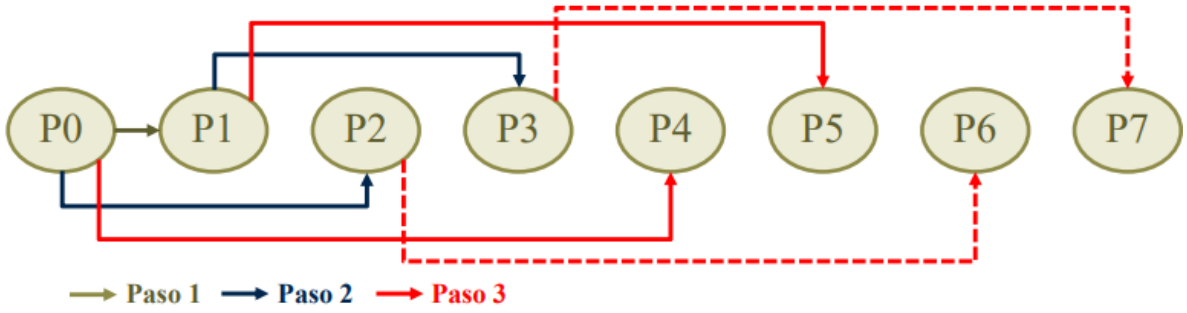
Comodines en Origen y Tag.

. 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, Otras…

Ejemplo: El proceso 0 tiene información y debe comunicar a los demás procesos. Se realiza un broadcast efectivo, y en tan solo 3 unidades de tiempo se transmite toda la información:



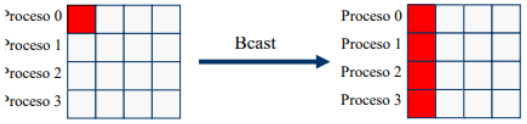
Sincronización en una barrera:

MPI_Barrier(MPI_Comm comunicador)

Broadcast:

Un proceso envía el mismo mensaje a todos los otros procesos (incluso a él) del comunicador:

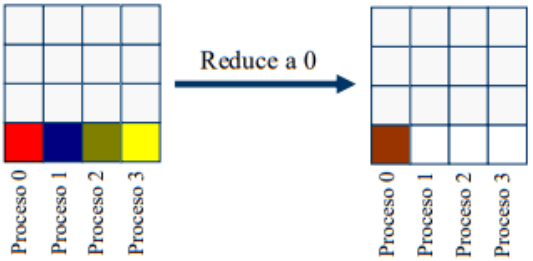
*MPI_Bcast (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, MPI_Comm comunicador)*



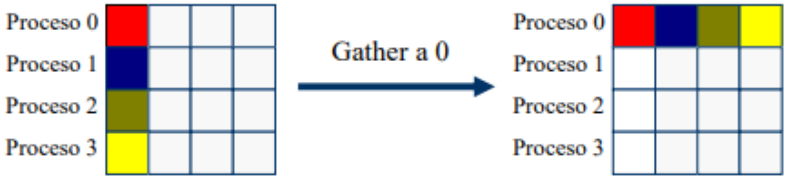
Reducción de todos a uno:

Combina los elementos enviados por cada uno de los procesos (inclusive el destino) aplicando una cierta operación:

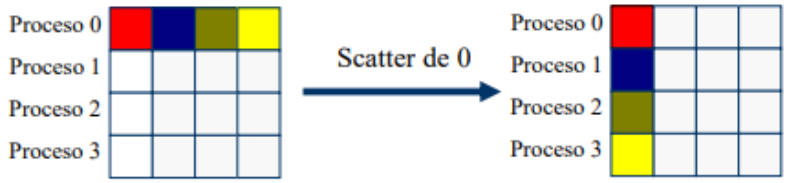
*MPI_Reduce (void *sendbuf, void *recvbuf, int cantidad, MPI_Datatype tipoDato, MPI_Op operación, int destino , MPI_Comm comunicador)*



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:
• Todos los vectores tienen igual tamaño.:
*MPI_Gather (void *sendbuf, int cantEnvio, MPI_Datatype, tipoDatoEnvio, void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int destino, MPI_Comm comunicador)*
• Los vectores pueden tener diferente tamaño:
*MPI_Gatherv (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int *cantsRec, int *desplazamientos, MPI_Datatype tipoDatoRec, int destino, MPI_Comm comunicador)*



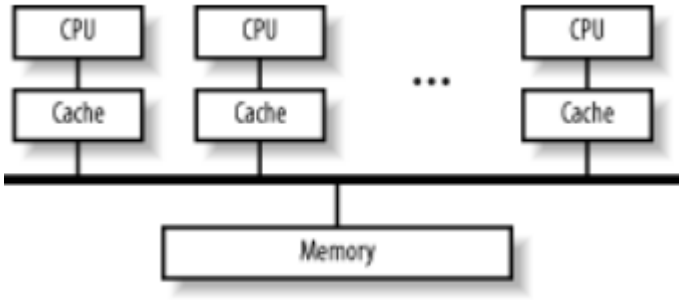
Scatter:
Reparte un vector de datos entre todos los procesos (inclusive el mismo dueño del vector):
• Reparte en forma equitativa (a todos la misma cantidad):
*MPI_Scatter (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int origen, MPI_Comm comunicador)*
• Puede darle a cada proceso diferente cantidad de elementos:
*MPI_Scatterv (void *sendbuf, int *cantsEnvio, int *desplazamientos, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int origen, MPI_Comm comunicador)*



Clase 10 | Programación Paralela

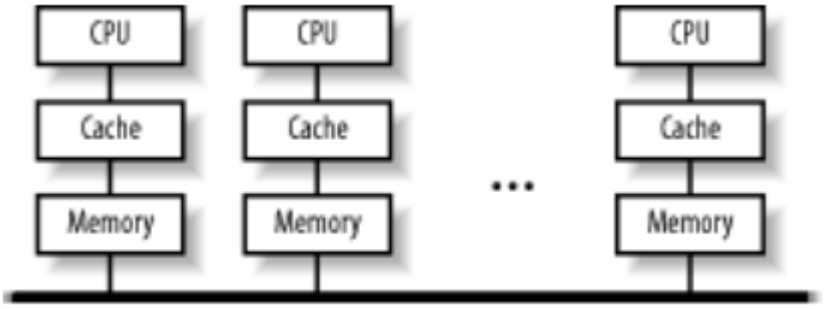
CLASIFICACIÓN DE ARQUITECTURAS PARALELAS

. Por la Organización del Espacio de Direcciones:
Memoria Compartida: Este enfoque utiliza accesos a memoria compartida, y se puede ver en multiprocesadores de memoria compartida donde hay interacción modificando datos sobre la misma memoria.
Podemos tener un esquema UMA (Uniform Memory Access) con bus o crossbar switch, pero que presenta problemas de sincronización y consistencia:



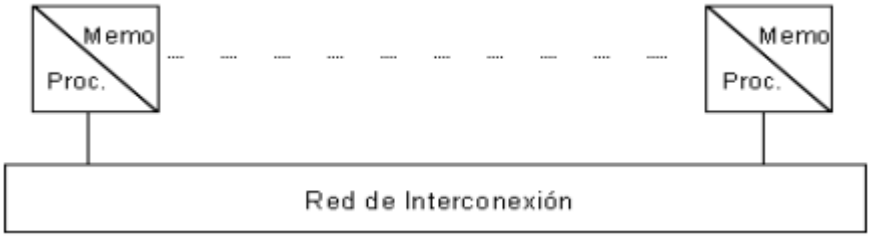
Esquema UMA

O podemos tener un esquema NUMA (Non Uniform Memory Access) para mayor número de procesadores distribuidos:



Esquema NUMA

Memoria Distribuida: Este enfoque utiliza intercambios de mensajes, que se puede ver con procesadores conectados por una red cada uno trabajando y accediendo a su memoria local (no hay problemas de consistencia) y con interacción por pasaje de mensajes:



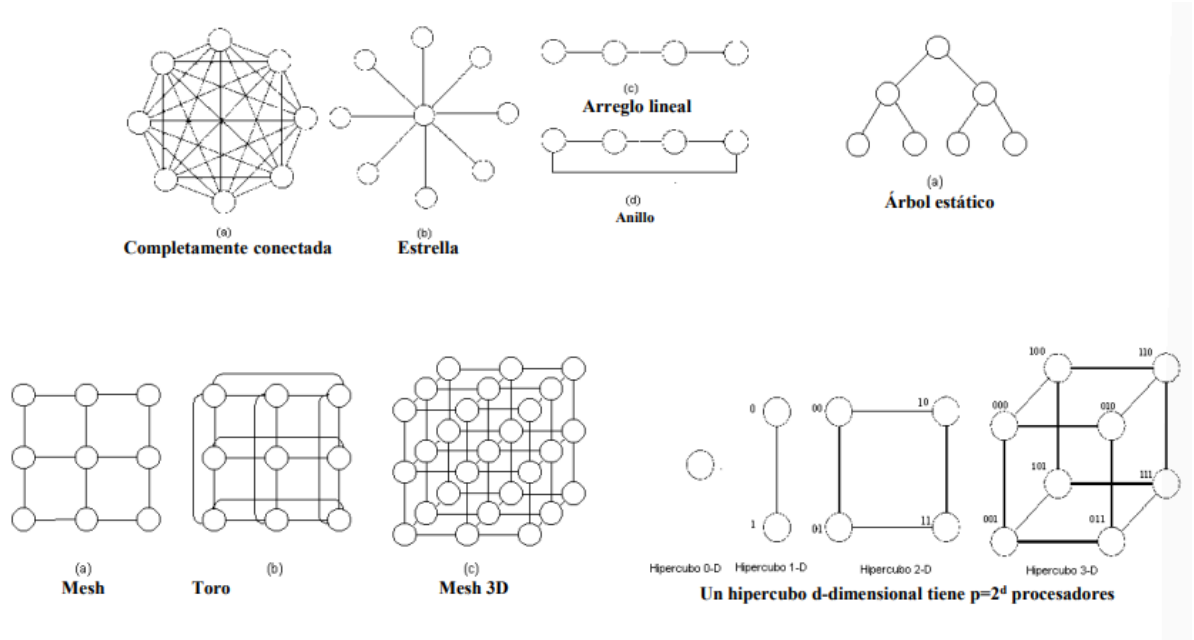
. Por la Granularidad:
Esta clasificación tiene que ver con la relación entre la capacidad de procesamiento, la cantidad de datos a procesar, y la distribución de los distintos nodos y memorias.

. Por el Mecanismo de Control:
Se basa en la manera en que las instrucciones son ejecutadas sobre los datos.
Single Instruction Single Data (SISD): Una sola instrucción a la vez afecta a una sola memoria. Las instrucciones son ejecutadas en secuencia, una por ciclo de instrucción.
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.
La ejecución es determinística.
Multiple Instruction Single Data (MISD): distintas instrucciones actúan sobre datos comunes. La operación es de forma sincrónica.
Ejemplo: Múltiples filtros de frecuencia operando sobre una única señal, o múltiples algoritmos de criptografía intentando crackear un único mensaje codificado.

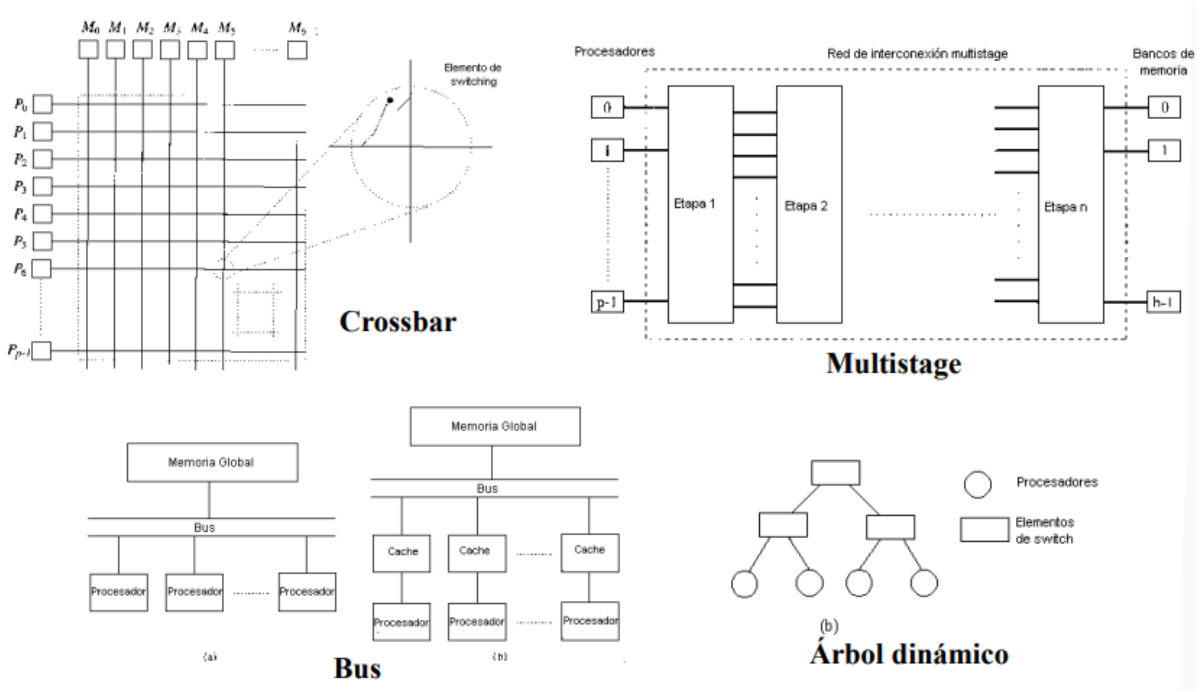
Single Instruction Multiple Data (SIMD): Conjunto de procesadores idénticos, cada uno con su memoria, que ejecutan la misma instrucción sobre distintos datos.
El host hace broadcast de la instrucción. Y la ejecución es sincrónica y determinística.
Multiple Instruction Multiple Data (MIMD): Cada procesador tiene su propio flujo de instrucciones y de datos, y 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).

. Por la Red de Interconexión:
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.).
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.



DISEÑO DE ALGORITMOS PARALELOS

. ¿Por qué es compleja la programación paralela?

- Decidir cuál es la granularidad óptima de las tareas: tareas más grandes con poca comunicación / tareas pequeñas con mucha comunicación.
- 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 (nodos con mucha carga y otros con poca).
- 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.

. Pasos para diseñar un algoritmo paralelo:

- Primero identificar porciones de trabajo (tareas) concurrentes. **FUNDAMENTAL.**
- Luego mapear tareas a procesos en distintos procesadores. **FUNDAMENTAL.**
- Distribuir datos de entrada, intermedios y de salida.
- Manejar el acceso a datos compartidos.
- Por último, sincronizar procesos.

. Descomposición en Tareas:

Para desarrollar un algoritmo paralelo el primer punto es descomponer el problema en sus componentes funcionales concurrentes (procesos/tareas).
En este paso 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.

Luego, en etapas posteriores, la evaluación de los requerimientos de comunicación, arquitectura de destino, o temas de IS pueden llevar a descartar algunas posibilidades detectadas en esta etapa, revisando la partición original y aglomerando tareas para incrementar su tamaño o granularidad.
Esta descomposición puede realizarse de muchos modos. Un primer concepto es pensar en tareas de igual código (normalmente paralelismo de datos o dominio) pero también podemos tener diferente código (paralelismo funcional).
En este paso se realiza también la **descomposición de datos**, que consta de 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.
Y se realiza también la **descomposición funcional**, donde primero se 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:
El algoritmo resultante de las etapas anteriores es abstracto en el sentido de que no es especializado para ejecución eficiente en una máquina particular.
Por eso, en la etapa de aglomeración se revisan las decisiones tomadas con la visión de obtener un algoritmo que ejecute en forma eficiente en una clase de máquina real.
En particular, se considera si es útil combinar o aglomerar las tareas para obtener otras de mayor tamaño. También se define si vale la pena replicar datos y/o computación.
Esta etapa tiene 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:
En esta etapa se especifica dónde se 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:

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.

Debe encontrarse un equilibrio que optimice el rendimiento paralelo: *EL MAPPING DETERMINA LA EFICIENCIA DEL ALGORITMO*.

MÉTRICAS DEL PARALELISMO

. En el mundo serial la performance con frecuencia es medida teniendo en cuenta los requerimientos de tiempo y memoria de un programa.
. Mientras que en un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance. Aquí hay también 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, es decir: sistema paralelo = algoritmo + arquitectura sobre la que se implementa.
. En la medición de performance es usual elegir un problema y testear el tiempo variando el 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) :
Es una medida de la mejora de rendimiento (performance) de una aplicación al agregar procesadores comparado con el rendimiento de usar un solo procesador.

S es el cociente entre el tiempo de ejecución del algoritmo serial conocido más rápido (Ts) y el tiempo de ejecución paralelo del algoritmo elegido (Tp):

$$S = \frac{T_S}{T_P}$$

El speedup óptimo depende de la arquitectura.
El speedup puede ser lineal o perfecto, sublineal y superlineal.

. Eficiencia (E) :
Es una medida relativa para la comparación de desempeños en diferentes entornos de computación paralela.
E es el cociente entre Speedup y Speedup Óptimo:

$$E = \frac{S}{S_{óptimo}}$$

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.

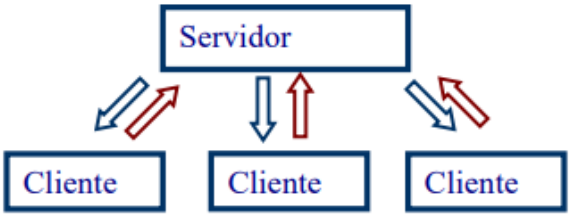
. Limitantes del Speedup:
• Alto porcentaje de código secuencial (Ley de Amdahl).
• Alto porcentaje de entrada/salida respecto de la computación.
• Algoritmo no adecuado (necesidad de rediseñar).
• Excesiva contención de memoria (rediseñar código para localidad de datos).
• Tamaño del problema (puede 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.

. Ley de Amdahl:
La Ley de Amdahl dice que para todo problema dado existe un máximo speedup alcanzable independientemente del número de procesadores que se utilicen. Esto significa que es el algoritmo quién decide la mejora de velocidad dependiendo de la cantidad de código no paralelizable, y no de la cantidad de procesadores, llegando finalmente a un momento donde no se puede paralelizar más el algoritmo.

PARADIGMAS DE PROGRAMACIÓN PARALELA

Un paradigma de programación es una clase de algoritmos que resuelve distintos problemas, pero tienen la misma estructura de control. Para cada paradigma puede escribirse un esqueleto algorítmico que define la estructura de control común.

. Cliente/Servidor:
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. • Mecanismos de invocación variados (rendezvous, RPC, monitores).



Cliente/Servidor

. Master/Worker:

El master envía iterativamente datos a los workers y recibe resultados de éstos. Puede ocurrir un posible "cuello de botella" (por ejemplo, por tareas muy chicas o workers 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).

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.

Pipe

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.

. SMPD:

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) Determina 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 tienen 1 o más loops. Cada proceso es un programa iterativo. Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones.

¿Qué sucede si hay menos de n procesadores?

Se puede dividir la matriz resultado en strips (subconjuntos de filas o columnas) y usar un proceso worker por strip.

El tamaño del strip óptimo es un problema interesante para balancear costo de procesamiento con costo de comunicaciones.