

# Programación concurrente

## Resumen teorías

<b>Clase 1 - Introducción</b>	<b>7</b>
Concurrencia	7
Objetivos de los sistemas concurrentes	7
Programa Secuencial	7
Posibles comportamientos de los procesos	8
Procesamiento secuencial, concurrente y paralelo	8
Secuencial	8
Paralelo	8
Concurrente	8
Procesos e hilos	10
Comunicación entre procesos	10
Sincronización entre procesos	11
Interferencia	11
Prioridad	12
Granularidad	12
Manejo de los recursos	13
Deadlock	13
Requerimientos para un lenguaje concurrente	14
Problemas asociados con la programación concurrente	14
Concurrencia y paralelismo	14
Concurrencia a nivel hardware	15
Niveles de memoria	16
Multiprocesadores de memoria compartida	16
Multiprocesadores con memoria distribuida	17
<b>Clase 2 - Acciones atómicas y sincronización - Propiedades y fairness</b>	<b>18</b>
Estado de un programa concurrente	18
Atomicidad de grano fino	18
“A lo sumo una vez”	19
Sincronización	19
Propiedades	20
Fairness y políticas de scheduling	21
Clasificación de las políticas de scheduling	21
Fairness incondicional	21
Fairness débil	21
Fairness fuerte	22
<b>Clase 3 - Variables compartidas</b>	<b>23</b>
Locks y barreras	23
Problema de la sección crítica	23
Propiedades a cumplir	24

Implementación de la sentencia await	24
Solución por hardware	25
Solución de “grano grueso”	25
Solución de “grano fino”: spin locks	27
Solución fair	28
Algoritmo tie-breaker (2 procesos)	28
Solución de “grano grueso” al algoritmo tie-breaker	29
Solución de “grano fino” al algoritmo tie-breaker	30
Algoritmo ticket	30
Solución de “grano grueso” al algoritmo ticket	31
Solución de “grano fino” al algoritmo ticket	32
Algoritmo bakery	32
Solución de “grano grueso” al algoritmo bakery	33
Solución de “grano fino” al algoritmo bakery	33
Problema de la sincronización de barrera	34
Solución con un contador compartido	34
Solución con flags y coordinadores	35
Solución de “grano grueso”	35
Solución de “grano fino”	36
Solución con árboles	37
Solución con barrera simétrica	37
Butterfly barrier	37
Defectos de la sincronización por busy waiting	38
<b>Clase 4 - Semáforos</b>	<b>39</b>
Defectos de la sincronización por busy waiting	39
Semáforos	39
Semáforo general (o counting semaphore)	39
Semáforo binario	40
Sección crítica: exclusión mutua	40
Barreras: señalización de eventos	40
Semáforos binarios divididos	41
Productores y consumidores con SBS	42
Buffers limitados: contadores de recursos	42
Un productor y un consumidor	42
Múltiples productores y múltiples consumidores	44
Varios procesos compitiendo por varios recursos compartidos	45
Problema de los filósofos: exclusión mutua selectiva	45
Lectores y escritores	46
Solución con exclusión mutua	47
Solución con sincronización por condición	48
Passing the baton	48
Alocación de recursos y scheduling	50
Alocación Shortest Job Next (SJN)	51
<b>Clase 5 - Monitores</b>	<b>54</b>

Problemas con los semáforos	54
Monitores	54
Exclusión mutua	55
Sincronización por condición	55
Programa concurrente	55
Notación	55
Sincronización	56
Operaciones adicionales que NO SON USADAS EN LA PRÁCTICA sobre las variables condición	57
Wait/signal vs P/V	58
<b>Clase 6 - Memoria distribuida y pasaje de mensajes asincrónico</b>	<b>59</b>
Características	59
Mecanismos para el Procesamiento Distribuido:	60
Patrones de interacción:	60
Pasaje de mensajes asincrónico (PMA)	61
Características de los canales	61
Productores y consumidores (filtro)	62
Cliente servidor	64
1 operación	65
Múltiples operaciones	66
Múltiples operaciones y variables de condición	68
Dualidad entre Monitores y Pasaje de Mensajes	69
Sentencias de alternativa múltiples	70
Continuidad conversacional	70
Pares (peers) interactuantes	72
Solución centralizada	72
Solución simétrica	73
Solución anillo circular	73
Análisis sobre las soluciones	74
<b>Clase 7 - Pasaje de mensajes sincrónico y paradigmas de interacción entre procesos</b>	<b>75</b>
Productor/consumidor	75
Deadlock en PMS	77
CSP	78
Ejemplos básicos	79
Comunicación guardada	79
Ejemplos	81
Copiar con buffer limitado	82
Asignación de recursos	82
Intercambio de valores	83
La Criba de Eratóstenes para generar números primos	83
Ordenación de un arreglo	84
Paradigmas de interacción entre procesos	87
Manager/worker	87

Heartbeat	88
Topología de una red	89
Pipeline	90
Probe-echo	91
Broadcast	91
Token passing	91
Servidores replicados	92
<b>Clase 8 - RPC y rendezvous</b>	<b>93</b>
Diferencias entre RPC y rendezvous	93
RPC	93
Sincronización de módulos	95
Ejemplo cliente/servidor	96
Time server	96
Manejo de caches en un sistema de archivos distribuido	96
Pares interactuantes	98
RPC en Java (Remote Method Invocation: RMI)	99
Rendezvous	99
Ejemplos	101
Buffer limitado	101
Filósofos centralizado	102
Time server	102
Alocador SJN	103
Rendezvous en ADA	103
Tasks	104
Sincronización: Call	104
Sincronización: Accept	105
Ejemplos	106
Mailbox para 1 mensaje	106
Lectores-escritores	107
Mailbox para N mensajes (buffer limitado)	108
Con una cola:	108
Con un arreglo:	108
Filósofos centralizado	109
Múltiples entry	109
Encolar pedidos:	110
Time server	111
Alocador SJN	112
<b>Clase 9 - Pthreads y MPI</b>	<b>113</b>
Pthreads	113
Creación de un hilo	113
Terminación de un hilo	113
Esperar a que un hilo termine	113
Cancelar un hilo	114
Exclusión mutua	114

Productores/Consumidores	114
Tipos de mutex	116
Overhead de bloqueos por exclusión mutua	117
VARIABLES DE CONDICIÓN	117
Productores/Consumidores	118
Atributos	119
Semáforos	120
Declaración y operaciones	120
Productor consumidor	120
Monitores	121
Lectores y escritores	122
MPI	123
Send y receive	123
Comunicación bloqueante	124
Comunicación no bloqueante	124
Inicio y finalización en MPI	124
Comunicadores	125
Adquisición de información	125
Tipos de datos para las comunicaciones	126
Comunicación punto a punto	126
Bloqueante	127
No bloqueante	128
Consulta de mensajes pendientes	130
Comunicaciones colectivas	130
Ejemplo de broadcast	131
MPI reduce	131
MPI gather	131
MPI scatter	132
Minimizar los overheads de comunicación	133
<b>Clase 10 - Introducción a la programación paralela</b>	<b>134</b>
Clasificación de arquitecturas paralelas	134
Por el espacio de direcciones	134
Multiprocesadores de memoria compartida	134
Multiprocesadores con memoria distribuida	134
Por granularidad	135
Por el mecanismo de control	135
SISD	136
SIMD	136
MISD	137
MIMD	137
Por la red de interconexión	138
Redes de interconexión dinámicas	139
Redes de interconexión estáticas	139
Diseño de algoritmos paralelos	140

¿Por qué es compleja la programación paralela?	140
Pasos a seguir	140
Descomposición en tareas	141
Descomposición de datos	141
Descomposición funcional	142
Aglomeración	142
Características de las tareas	142
Mapeo de tareas a procesadores	143
Métricas de rendimiento	144
Speedup (S)	144
Eficiencia	145
Escalabilidad	146
Factores que limitan el speedup	146
Paradigmas de programación paralela	146
Cliente/servidor	147
Master/slave o master/worker	147
Pipeline y Algoritmos Sistólicos	148
Dividir y conquistar	148
Problema de la cuadratura	148
SPMD	150
Multiplicación de matrices	151

# Clase 1 - Introducción

## Concurrencia

Es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente

Permite a distintos objetos actuar al mismo tiempo

Factor relevante para el diseño de hardware, sistemas operativos, multiprocesadores, computación distribuida, programación y diseño

No hay un orden preestablecido en la ejecución => no determinismo (ejecuciones con la misma “entrada” puede generar diferentes “salidas”)

## Objetivos de los sistemas concurrentes

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

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

Algunas ventajas:

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

## Programa Secuencial

Un único flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente

Por ahora llamaremos “Proceso” a un programa secuencial

Un único hilo o flujo de control → programación secuencial, monoprocesador

Múltiples hilos o flujos de control → programa concurrente o programa paralelo

## Posibles comportamientos de los procesos

- Procesos independientes: cada proceso tiene los recursos que necesita y actúa de forma independiente
- Competencia: es típico en sistemas operativos y redes, debido a recursos compartidos
- Cooperación:
  - Los procesos se combinan para resolver una tarea común
  - Hay que tener en cuenta la sincronización

## Procesamiento secuencial, concurrente y paralelo

### Secuencial

Pasos realizados uno después del otro. Debe haber un estricto orden temporal

### Paralelo

Pasos realizados a la vez

Consecuencias:

- Menor tiempo para completar el trabajo
- Menor esfuerzo individual
- Paralelismo del hardware

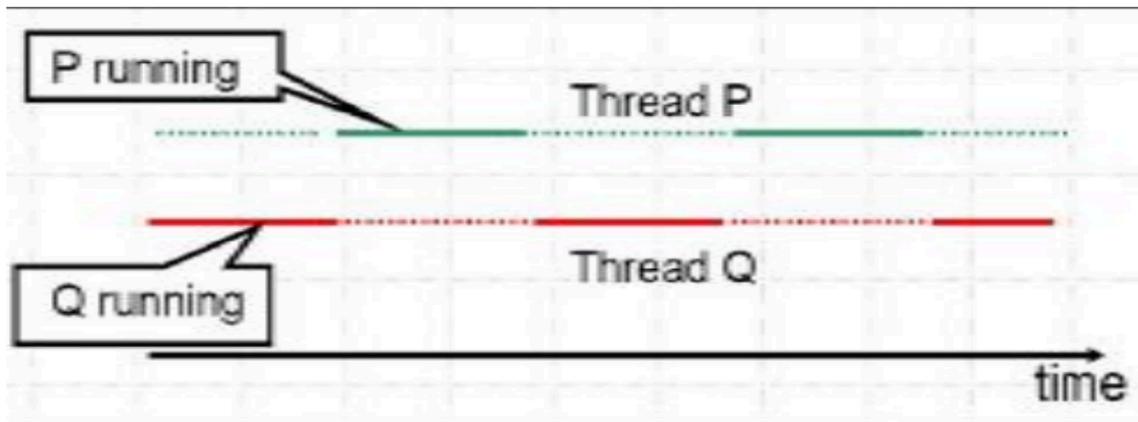
Dificultades:

- Distribución de la carga de trabajo
- Necesidad de compartir recursos evitando conflictos
- Necesidad de esperarse en puntos clave
- Necesidad de comunicarse
- Tratamiento de las fallas

### Concurrente

Pasos alternados entre los diferentes procesos

Concurrencia sin paralelismo de hardware



Dificultades:

- Distribución de carga de trabajo
- Necesidad de compartir recursos
- Necesidad de esperarse en puntos clave
- Necesidad de comunicarse
- Necesidad de recuperar el “estado” de cada proceso al retomarlo

La concurrencia es un concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores

Este último caso sería multiprogramación en un procesador:

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

Un programa concurrente especifica dos o más “programas secuenciales” que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos

Un proceso o tarea es un elemento concurrente abstracto que puede ejecutarse simultáneamente con otros procesos o tareas, si el hardware lo permite (por ejemplo los TASKs de ADA)

Un programa concurrente puede tener N procesos habilitados para ejecutarse concurrentemente y un sistema concurrente puede disponer de M procesadores cada uno de los cuales puede ejecutar uno o más procesos

Características importantes:

- Interacción
- No determinismo => dificultad para la interpretación y debug

## Procesos e hilos

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

Procesos livianos, threads o hilos:

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

## Comunicación entre procesos

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

Es la posesión de información acerca de otro proceso para coordinar actividades

Los procesos se sincronizan:

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

**Ejemplo 1:** nunca se debería dividir por 0.

```
int x, y, z;

process A1
{
  ....
  y = 0;
  ....
}

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

**Ejemplo 2:** siempre *Público* debería terminar con valor igual a  $E1+E2$ .

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

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

## Prioridad

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

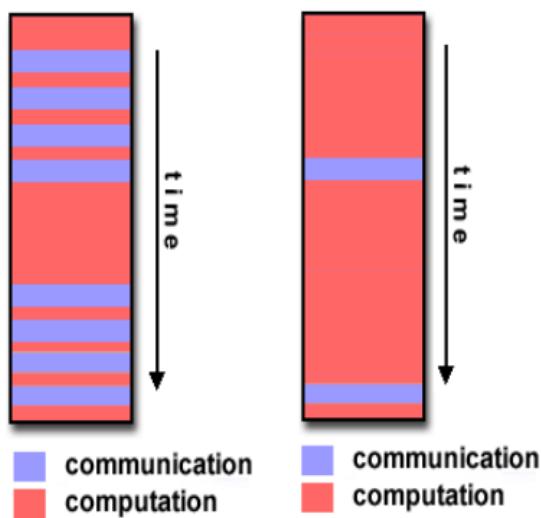
## Granularidad

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

Relación y adaptación a la arquitectura

Grano fino: cuando hay poco código y luego comunicación, una y otra vez

Grano grueso: cuando hay mucho código y luego comunicación, una y otra vez



Por ejemplo, en un clúster, conviene granularidad gruesa, porque la comunicación es costosa, pero el cómputo no

En un multicore conviene una solución de grano fino, porque tiene una comunicación veloz y un cómputo no tan bueno

## Manejo de los recursos

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

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

## Deadlock

Dos (o más) procesos pueden entrar en deadlock, si por error de programación ambos se quedan esperando que el otro libere un recurso compartido

La ausencia de deadlock es una propiedad necesaria en los procesos concurrentes

4 propiedades necesarias y suficientes para que exista deadlock son:

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

Con que una de las cuatro propiedades no esté presente, se garantiza la ausencia de deadlock

## Requerimientos para un lenguaje concurrente

Independientemente del mecanismo de comunicación/sincronización entre procesos, los lenguajes de programación concurrente deberán proveer primitivas adecuadas para la especificación e implementación de las mismas

Requerimientos de un lenguaje de programación concurrente:

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

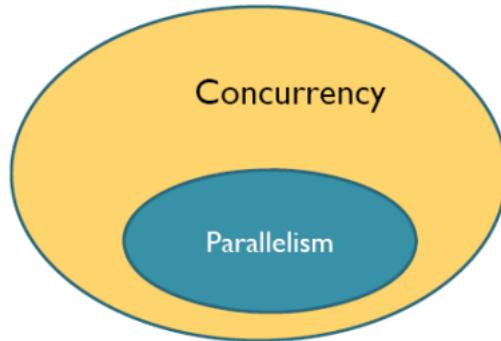
## Problemas asociados con la programación concurrente

- Los procesos no son independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas
- Los procesos iniciados dentro de un programa concurrente pueden NO estar “vivos”. Esta pérdida de la propiedad de liveness puede indicar deadlocks o una mala distribución de recursos
- Hay un no determinismo implícito en el interleaving de procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas dificultad para la interpretación y debug
- Posible reducción de performance por overhead de context switch, comunicación, sincronización, etc.
- 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 y paralelismo

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

Paralelismo: se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución



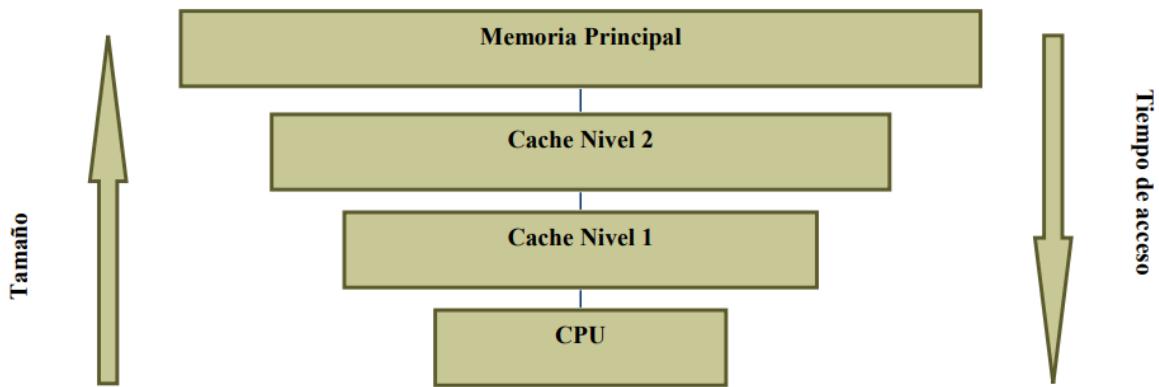
## Concurrencia a nivel hardware

Límite físico en la velocidad de los procesadores:

1. Máquinas monoprocesador ya no pueden mejorar
2. Más procesadores por chip para mayor potencia de cómputo
3. Multicores
4. Clúster de multicores:
  - Esto eleva el consumo energético
  - Se hace necesario mezclar memoria compartida (entre los cores de un multicore) con pasaje de mensajes (entre los multicores del clúster)

Para hacer un uso eficiente de estas nuevas arquitecturas se debe realizar una programación concurrente y paralela

## Niveles de memoria



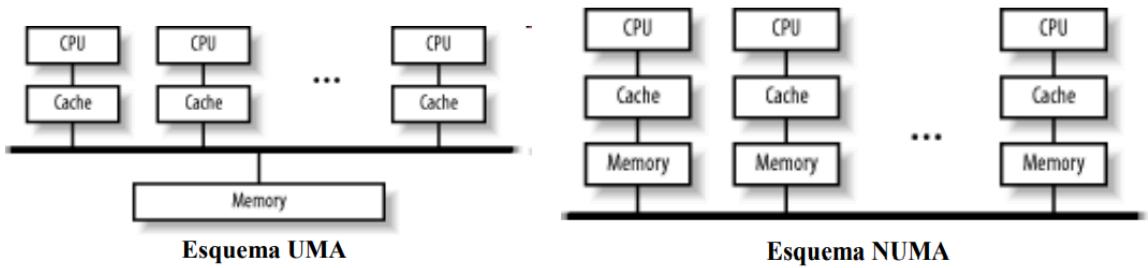
Al tener memoria compartida, las cachés no van a ser completamente aprovechadas porque cada proceso va a tener sus propios datos, lo cual va a romper con el principio de localidad temporal y espacial de los datos

Se debe tener un protocolo de consistencia de caché, para de esta manera, trabajar con datos consistentes, a pesar de que hayan sido modificados en cachés de otros procesadores

## 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
  - El acceso a memoria es no uniforme porque si debo acceder a una posición de memoria que está en mí memoria es más rápido que acceder a una posición de memoria en la memoria de otro procesador
  - No es memoria distribuida, ya que los procesadores con sus memorias están dentro de la misma máquina
- Problema de consistencia mencionado en el punto anterior



## Multiprocesadores con memoria distribuida

Procesadores conectados por una red

Memoria local (no hay problemas de consistencia)

Se debe usar la comunicación por pasaje de mensajes

Grado de acoplamiento de los procesadores:

- Multicomputadores (máquinas fuertemente acopladas):
  - Procesadores y red físicamente cerca
  - Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores
  - Alto ancho de banda y velocidad
- Memoria compartida distribuida
- Clusters
- Redes (multiprocesador débilmente acoplado)

# Clase 2 - Acciones atómicas y sincronización -

## Propiedades y fairness

### Estado de un programa concurrente

Valor que tienen las variables compartidas, locales, de los registros, etc. en un cierto instante de tiempo

### Atomicidad de grano fino

Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas

Una acción atómica hace una transformación de estados indivisibles (estados intermedios invisibles para otros procesos)

La ejecución de un programa concurrente consiste en un intercalado (interleaving) de las acciones atómicas ejecutadas por procesos individuales

Historia de un programa concurrente (trace): ejecución de un programa concurrente con un interleaving particular. En general el número de posibles historias de un programa concurrente es enorme; pero no todas son válidas

La interacción entre los procesos determina cuáles historias son correctas

Se debe asegurar un orden temporal entre las acciones que ejecutan los procesos  
Como las tareas se intercalan, deben fijarse restricciones

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

Una acción atómica de grano fino (fine grained) se debe implementar por hardware  
¿La operación de asignación  $A=B$  es atómica? No, porque implica:

1. Load PosMemB, reg
2. Store reg, PosMemA

Si una expresión  $e$  en un proceso no referencia una variable alterada por otro proceso, la evaluación será atómica, aunque requiera ejecutar varias acciones atómicas de grano fino

Si una asignación  $x = e$  en un proceso no referencia ninguna variable alterada por otro proceso, la ejecución de la asignación será atómica

Referencia crítica en una expresión es una referencia a una variable que es modificada por otro proceso

## “A lo sumo una vez”

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

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

Una 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

## Sincronización

Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente

En general, es necesario ejecutar secuencias de sentencias como una única acción atómica (hace falta sincronización por exclusión mutua)

Necesitamos un mecanismo de sincronización para construir una acción atómica de grano grueso (coarse grained) como secuencia de acciones atómicas de grano fino (fine grained) que aparecen como indivisibles

El siguiente es un caso de sincronización por exclusión mutua donde, cuando un proceso comienza a ejecutar la expresión atómica de grano grueso  $e$ , ningún otro proceso podrá ejecutar nada de esa expresión:

- $\langle e \rangle$  indica que la expresión  $e$  debe ser evaluada atómicamente
- Llamada sentencia atómica incondicional

El siguiente es un caso de sincronización por exclusión mutua junto con sincronización por condición:

- <await (B) S;> se utiliza para especificar sincronización
- La expresión booleana B especifica una condición de demora
- S es una secuencia de sentencias que se garantiza que termina
- Se garantiza que B es true cuando comienza la ejecución de S
- Ningún estado interno de S es visible para los otros procesos
- Llamada sentencia atómica condicional
- Podría hacerse <await (B)>, haciendo uso únicamente de sincronización por condición:
  - Si B satisface ASV, puede implementarse como busy waiting o spin loop:
    - do (not B) → skip od
    - while (not B) → skip elihw;

## Propiedades

Una propiedad de un programa concurrente es un atributo verdadero en cualquiera de las historias de ejecución del mismo

Toda propiedad puede ser formulada en términos de dos clases: seguridad y vida

- Seguridad (safety) (algo malo nunca sucede):
  - Son características que nos aseguran que nada malo le ocurre a un proceso: asegura estados consistentes
  - Una falla de seguridad indica que algo anda mal
  - Ejemplos de propiedades de seguridad: exclusión mutua, ausencia de interferencia entre procesos, partial correctness (siempre que un programa termina, lo hará correctamente)
- Vida (liveness) (algo bueno eventualmente sucederá):
  - Son características que nos aseguran que eventualmente ocurre algo bueno con una actividad: progresá, no hay deadlocks
  - Una falla de vida indica que las cosas dejan de ejecutar
  - Ejemplos de vida: terminación (asegura que el programa va a terminar), asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, ausencia de inanición (apetito 😊), ausencia de deadlock, etc. => dependen de las políticas de scheduling

¿Qué pasa con la total correctness? => Nos asegura que un programa va a terminar (terminación) y que lo hará correctamente (partial correctness)

## Fairness y políticas de scheduling

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

Una acción atómica en un proceso es elegible si es la próxima acción atómica en el proceso que será ejecutada

Si hay varios procesos => hay varias acciones atómicas elegibles

Una política de scheduling determina cuál será la próxima en ejecutarse

Ejemplo: si la política es asignar un procesador a un proceso hasta que termina o se demora

¿Qué podría suceder en este caso?:

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

## Clasificación de las políticas de scheduling

### Fairness incondicional

Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional (que no tiene await) que es elegible eventualmente es ejecutada.

Round robin es incondicionalmente fair en monoprocesador, y la ejecución paralela lo es en un multiprocesador

No nos asegura que, en caso de encontrar una sentencia atómica condicional, vaya a continuar la correcta ejecución del programa

### Fairness débil

Una política de scheduling es débilmente fair si:

1. Es incondicionalmente fair y
2. Toda acción atómica condicional (con await) que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece true hasta que es vista por el proceso que ejecuta la acción atómica condicional

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

### Fairness fuerte

Una política de scheduling es fuertemente fair si:

1. Es incondicionalmente fair y
2. Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia

Ejemplo: ¿Este programa termina?

```
bool continue = true, try = false;
co
  while (continue) {
    try = true;
    try = false;
  }
  // <await (try) continue = false>
oc
```

No es simple tener una política que sea práctica y fuertemente fair

En el ejemplo anterior, con 1 procesador, una política que alterna las acciones de los procesos sería fuertemente fair, pero es impráctica

Round-robin es práctica pero no es fuertemente fair (es débilmente fair)

# Clase 3 - Variables compartidas

## Locks y barreras

Hay dos problemas a solucionar:

1. Problema de la sección crítica: implementación de acciones atómicas por software (de grano grueso)
2. Sincronización de barrera: punto de sincronización que todos los procesos deben alcanzar para que cualquier proceso pueda continuar

Lo que vamos a usar para resolver estos problemas es la técnica de busy waiting

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

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

## Problema de la sección crítica

Se puede resolver aplicando protocolos de entrada y salida a dichas acciones críticas (locks)

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

Las soluciones a este problema pueden usarse para implementar sentencias await arbitrarias (incondicionales): <SC>

## Propiedades a cumplir

- 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 y tiene posibilidades de hacerlo (eventualmente lo hará)

Solución trivial: <SC>. Pero, ¿cómo se implementan los <>?

## Implementación de la sentencia await

Cualquier solución al problema de la SC se puede usar para implementar una acción atómica incondicional <S;> =>

```
SCEnter;  
S;  
SCExit
```

Para una acción atómica condicional <await (B) S;> =>

```
SCEnter;  
while (not B) {  
    SCExit;  
    SCEnter;  
}  
S;  
SCExit;
```

Si S es skip, y B cumple ASV, await (B); puede implementarse por medio de while (not B) skip;

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

Para reducir contención de memoria =>

```
SCEnter;
```

```

while (not B) {
    SCExit;
    Delay;
    SCEnter;
}
S;
SCExit;

```

Solución 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 → penalización de performance

La solución no es correcta en un multiprocesador

Solución de “grano grueso”

<b>bool in1=false, in2=false # MUTEX: <math>\neg(in1 \wedge in2)</math> #</b>
---

<b>process</b> SC1     { <b>while</b> (true)         { <b>in1 = true;</b> <i># protocolo de entrada</i> <b>sección crítica;</b> <b>in1 = false;</b> <i># protocolo de salida</i> <b>sección no crítica;</b> } }
---

<b>process</b> SC2     { <b>while</b> (true)         { <b>in2 = true;</b> <i># protocolo de entrada</i> <b>sección crítica;</b> <b>in2 = false;</b> <i># protocolo de salida</i> <b>sección no crítica;</b> } }
---

No asegura la propiedad de exclusión mutua.

Podría darse que in1 e in2 se pongan en true y se ejecuten ambas secciones críticas a la vez 😱

Para que se cumpla la propiedad de exclusión mutua, in1 e in2 no pueden ser verdaderas a la vez

```
process SC1
{ while (true)
  {<await (not in2) in1 = true;>
    sección crítica;
    in1 = false;
    sección no crítica;
  }
}
```

```
process SC2
{ while (true)
  {<await (not in1) in2 = true;>
    sección crítica;
    in2 = false;
    sección no crítica;
  }
}
```

Propiedades a cumplir:

- Exclusión mutua: por construcción, SC1 y SC2 se excluyen en el acceso a la SC
- Ausencia de deadlock: si hay deadlock, SC1 y SC2 están bloqueados en su protocolo de entrada, entonces, in1 e in2 serían true a la vez. Esto no puede darse ya que ambas son falsas en ese punto (lo son inicialmente, y al salir de SC, cada proceso vuelve a serlo)
- Ausencia de demora innecesaria: si SC1 está fuera de su SC o terminó, in1 es false; si SC2 está tratando de entrar a SC y no puede, in1 es true; entonces, para que SC2 no pueda entrar a SC si SC1 está fuera de su SC o terminó, in1 tendría que ser false y true a la vez => no hay demora innecesaria
- Eventual entrada:
  - Si SC1 está tratando de entrar a su SC y no puede, SC2 está en SC (in2 es true). Un proceso que está en SC eventualmente sale → in2 será false y la guarda de SC1 true
  - Análogamente para SC2
  - Si los procesos corren en procesadores iguales y el tiempo de acceso a SC es finito, las guardas son true con infinita frecuencia
  - El problema viene cuando por ejemplo SC1 entra en la SC, entonces SC2 no puede entrar a la SC porque in1 es true, SC1 podría salir de SC, poner in1 en false, volver a entrar al loop y poner in1 en true antes de que SC2 tome el control

- Solo se puede garantizar la eventual entrada con una política de scheduling fuertemente fair

Para  $n$  procesos cambiamos las variables:

**bool lock=false; # lock = in1 v in2 #**

<b>process SC1</b> { while (true) { ⟨await (not lock) lock=true;⟩ sección crítica; lock = false; sección no crítica; }	<b>process SC2</b> { while (true) { ⟨await (not lock) lock=true;⟩ sección crítica; lock = false; sección no crítica; }
---	---

Entonces:

<b>process SC [i=1..n]</b> { while (true) { ⟨await (not lock) lock=true;⟩ sección crítica; lock = false; sección no crítica; }
---

Seguimos teniendo una solución de grano grueso (porque seguimos usando la sentencia await)

Solución de “grano fino”: spin locks

Objetivo: hacer “atómico” el await de grano grueso

Idea: usar instrucciones como test & set (TS), fetch & add (FA) o compare & swap, disponibles en la mayoría de los procesadores

Usamos test & set:

<b>bool TS (bool ok);</b> { < bool inicial = ok; ok = true; return inicial; > }
---

Setea la variable pasada como parámetro en true y retorna su valor inicial

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

```
bool lock=false;  
process SC[i=1 to n]  
{ while (true)  
  { while (TS(lock)) skip ;  
    sección crítica;  
    lock = false;  
    sección no crítica;  
  }
```



Cumple las 4 propiedades si el scheduling es fuertemente fair

Una política débilmente fair es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC)

TS escribe siempre en lock aunque el valor no cambie => Mejor test-and test-and-set

```
bool lock=false;  
process SC[i=1 to n]  
{ while (true)  
  { while (TS(lock)) skip ;  
    sección crítica;  
    lock = false;  
    sección no crítica;  
  }
```

```
while (lock) skip;  
while (TS(lock))  
  while (lock) skip;
```



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

## Solución fair

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

Algoritmo tie-breaker (2 procesos)

Protocolo de SC que requiere scheduling sólo débilmente fair y no usa instrucciones especiales => más complejo

Usa una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la sección crítica, y una variable adicional para

romper empates, indicando qué proceso fue el último en comenzar dicha entrada => esta última variable es compartida y de acceso protegido

Demora (quita prioridad) al último en comenzar su entry protocol

Solución de “grano grueso” al algoritmo tie-breaker

```
bool in1 = false, in2 = false;
int ultimo = 1;

process SC1 {
    while (true) {
        ultimo = 1; in1 = true;
        <await (not in2 or ultimo==2);>
        sección crítica;
        in1 = false;
        sección no crítica;
    }
}

process SC2 {
    while (true) {
        ultimo = 2; in2 = true;
        <await (not in1 or ultimo==1);>
        sección crítica;
        in2 = false;
        sección no crítica;
    }
}
```

Solución de “grano fino” al algoritmo tie-breaker

```
bool in1 = false, in2 = false;
int ultimo = 1;

process SC1 {
    while (true) {
        in1 = true; ultimo = 1;
        while (in2 and ultimo == 1) skip;
        sección crítica;
        in1 = false;
        sección no crítica;
    }
}

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

Generalización a n procesos:

- Si hay n procesos, el protocolo de entrada en cada uno es un loop que itera a través de n-1 etapas
- En cada etapa se usan instancias de tie-breaker para dos procesos para determinar cuáles avanzan a la siguiente etapa
- Si a lo sumo a un proceso a la vez se le permite ir por las n-1 etapas a lo sumo uno a la vez puede estar en la SC
- Es complejo y costoso en tiempo

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 hayan sido atendidos

Solución de “grano grueso” al algoritmo ticket

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

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

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

Potencial problema: los valores de próximo y turno son ilimitados. En la práctica, podrían resetearse a un valor chico (por ejemplo, 1)

Cumplimiento de las propiedades:

- El predicado TICKET es un invariante global, pues número es leído e incrementado en una acción atómica y próximo es incrementado en una acción atómica => hay a lo sumo un proceso en la SC
- La ausencia de deadlock y de demora innecesaria resultan de que los valores de turno son únicos
- Con scheduling débilmente fair se asegura eventual entrada

El await puede implementarse con busy waiting (la expresión booleana referencia una sola variable compartida)

El incremento de próximo puede ser un load/store normal (ya que el único proceso que puede modificar próximo es el que está en la SC y a lo sumo un proceso puede estar ejecutando su protocolo de salida)

Solución de “grano fino” al algoritmo ticket

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

¿Cómo se implementa la primera acción atómica donde se asigna el número?

Usando fetch and add: FA(var, incr): <temp=var; var=var+incr; return(temp)>

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

Si no existe FA se debe simular con una SC y la solución puede no ser fair

Algoritmo bakery

Cada proceso que trata de ingresar recorre los números de los demás y se auto asigna uno mayor

Luego espera a que su número sea el menor de los que esperan

Los procesos se chequean entre ellos y no contra un global

El algoritmo Bakery es más complejo, pero es fair y no requiere instrucciones especiales

No requiere un contador global próximo que se “entrega” a cada proceso al llegar a la SC

Solución de “grano grueso” al algoritmo bakery

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

{BAKERY: ( $\forall i: 1 \leq i \leq n: (SC[i] \text{ está en su SC}) \Rightarrow (turno[i] > 0) \wedge (\forall j: 1 \leq j \leq n, j \neq i: turno[j] = 0 \vee turno[i] < turno[j])$ )}

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

Esta solución de grano grueso no es implementable directamente:

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

Solución de “grano fino” al algoritmo bakery

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

{BAKERY: ( $\forall i: 1 \leq i \leq n: (SC[i] \text{ está en su SC}) \Rightarrow (turno[i] > 0) \wedge (\forall j: 1 \leq j \leq n, j \neq i: turno[j] = 0 \vee turno[i] < turno[j])$ )}

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
  }
}
```

## Problema de la sincronización de barrera

Una barrera es un punto de demora a la que deben llegar todos los procesos antes de permitirles pasar y continuar su ejecución

Dependiendo de la aplicación las barreras pueden necesitar reutilizarse más de una vez (por ejemplo en algoritmos iterativos)

### Solución con un contador compartido

n procesos necesitan encontrarse en una barrera:

- Cada proceso incrementa una variable cantidad al llegar
- Cuando cantidad es n, los procesos pueden pasar

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

Se puede implementar con:

```
FA(cantidad,1);
while (cantidad <> n) skip;
```

Problemas:

- El uso de la instrucción FA
- La implementación actual sirve para una sola iteración

## Solución con flags y coordinadores

Si no existe FA → puede distribuirse cantidad usando n variables (arreglo arribo[1..n])

El await pasaría a ser: <await(arribo[1] + ... + arribo[n] == n);>

Reintroduce contención de memoria y es ineficiente

Entonces, se hace uso de un proceso coordinador que, espera a que todos los procesos trabajadores arriben y le indica a cada uno que puede continuar

Solución de “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 de “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;
    }
}
```

Problemas:

- Requiere un proceso extra y, probablemente, un procesador extra, porque el coordinador debería trabajar continuamente sin que los saquen del procesador
- El tiempo de ejecución del coordinador es proporcional a  $n$

## Solución con árboles

Se combinan las acciones de workers y coordinador, haciendo que cada worker sea también coordinador

Por ejemplo, workers en forma de árbol: las señales de arriba van hacia arriba en el árbol, y las de continuar hacia abajo => combining tree barrier (más eficiente para n grande)

En combining tree barrier los procesos juegan diferentes roles

Problema: los diferentes procesos tienen distintos roles dependiendo de la posición del árbol donde estén (hoja, nodo intermedio o raíz), llevando a una multiplicación del código

## Solución con barrera simétrica

Una barrera simétrica para n procesos se construye a partir de pares de barreras simples para dos procesos:

<b>W[i]::</b> < await (arribo[i] == 0); > arribo[i] = 1; < await (arribo[j] == 1); > arribo[j] = 0;	<b>W[j]::</b> < await (arribo[j] == 0); > arribo[j] = 1; < await (arribo[i] == 1); > arribo[i] = 0;
--	--

Para construir una barrera de n procesos: usamos worker[1:n] arreglo de procesos

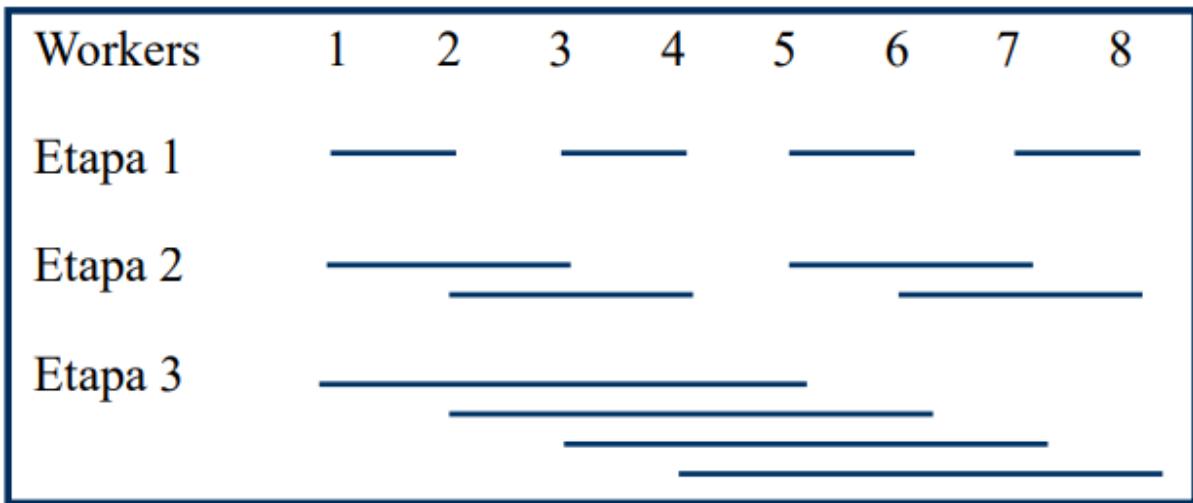
Si n es potencia de 2 podemos usar butterfly barrier

## Butterfly barrier

Tiene  $\log_2 n$  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  $\log_2 n$  etapas, todos pueden seguir



## Defectos de la sincronización por busy waiting

Protocolos “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

El procesador ve a un proceso que está esperando que se cumpla una guarda como que se está ejecutando, por lo que va a gastar todo su quantum chequeando una y otra vez la condición. De esta manera, retrasará la ejecución del programa en su totalidad

Se necesitan herramientas para diseñar protocolos de sincronización

# Clase 4 - Semáforos

## Defectos de la sincronización por busy waiting

Protocolos “busy-waiting”: complejos y sin clara separación entre variables de sincronización y las usadas para computar resultados

Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos

Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso spinning puede ser usado de manera más productiva por otro proceso

Necesidad de herramientas para diseñar protocolos de sincronización

Estos defectos implican una necesidad de herramientas para diseñar protocolos de sincronización

## Semáforos

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

Internamente el valor de un semáforo es un entero no negativo:

- V → Señala la ocurrencia de un evento (incrementa)
- P → Se usa para demorar un proceso hasta que ocurra un evento (decrementa)

Analogía con la sincronización del tránsito para evitar colisiones

Permiten proteger secciones críticas y pueden usarse para implementar sincronización por condición

## Semáforo general (o counting semaphore)

P(s): <await (s > 0) s = s-1;>

V(s): <s = s+1;>

Se indica que ha ocurrido un evento (V), cuando el semáforo es mayor que cero

## Semáforo binario

P(b): <await (b > 0) b = b-1;>

V(b): <await (b < 1) b = b+1;>

## Sección crítica: exclusión mutua

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

## Barreras: señalización de eventos

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

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

Semáforo de señalización: generalmente inicializado en 0

Un proceso señala el evento con V(s); otros procesos esperan la ocurrencia del evento ejecutando P(s)

```

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

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

```

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

## Semáforos binarios divididos

En la materia son semáforos generales que, por su forma de uso no van a adquirir un valor mayor que uno

Semáforo Binario Dividido (Split Binary Semaphore): los semáforos  $b_1, \dots, b_n$  forman un SBS en un programa si el siguiente es un invariante global:  $SPLIT: 0 \leq b_1 + \dots + b_n \leq 1$

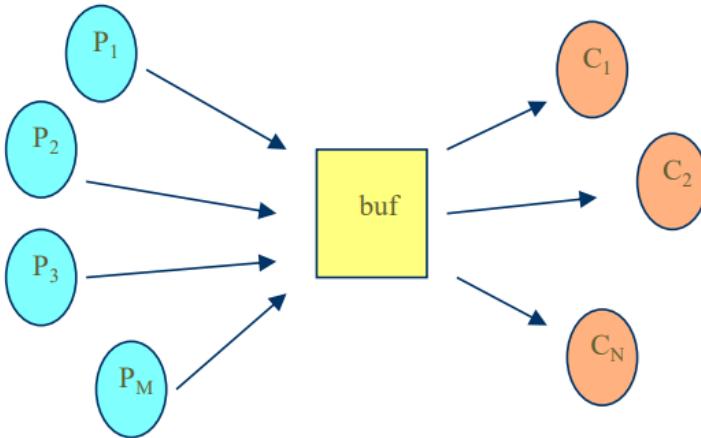
Es decir, solo uno de los semáforos  $b_i$  puede estar en 1

Los  $b_i$  pueden verse como un único semáforo binario  $b$  que fue dividido en  $n$  semáforos binarios

Son importantes por la forma en que pueden usarse para implementar mutex (en general la ejecución de los procesos inicia con un P sobre un semáforo y termina con un V sobre otro de ellos)

Las sentencias entre el P y el V se ejecutan con exclusión mutua

## Productores y consumidores con SBS



```
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
    ...
  }
}
```

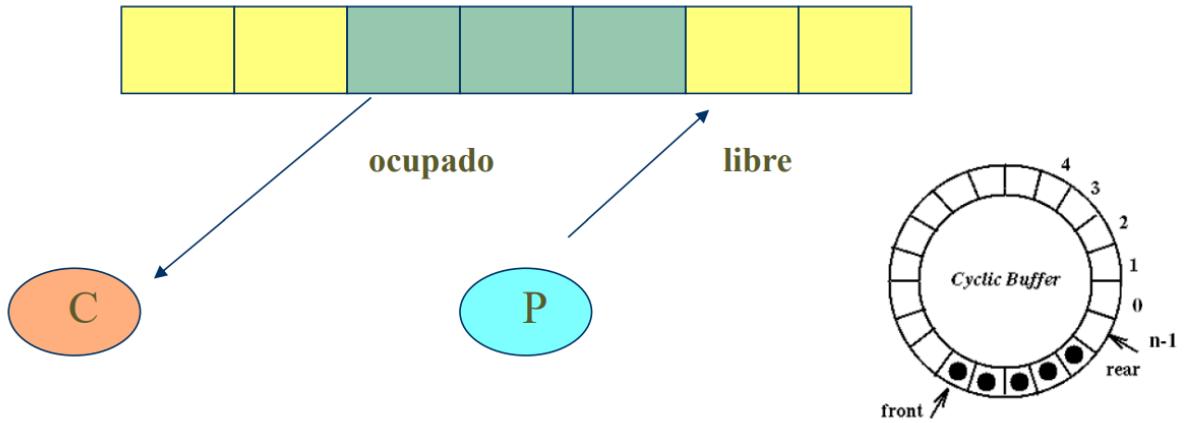
Los semáforos vacío y lleno forman un SBS

## Buffers limitados: contadores de recursos

Un productor y un consumidor

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



```

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
  }
}

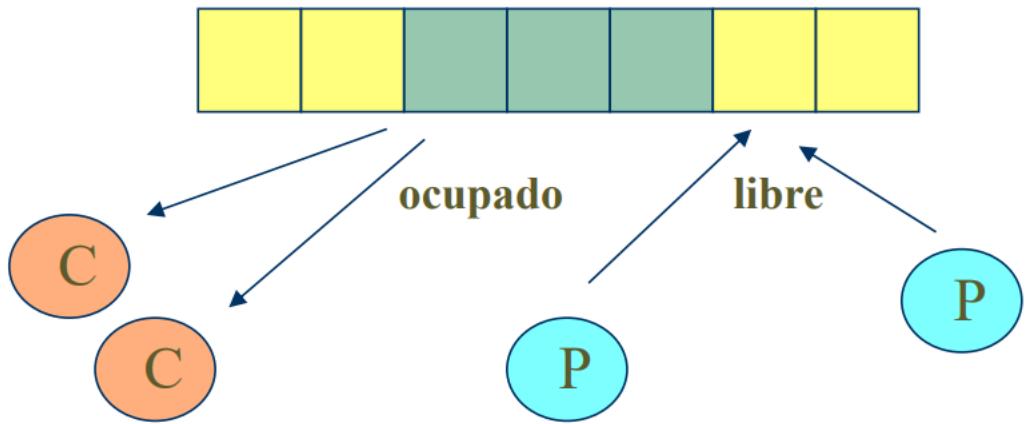
process Consumidor
{ while(true)
  { P(lleno); resultado = buf[ocupado]; ocupado = (ocupado+1) mod n; V(vacio); #retirar
    consumir mensaje resultado
    ...
  }
}

```

La variable vacio cuenta los lugares libres, y lleno los ocupados

Las acciones depositar y retirar se pudieron asumir atómicas pues sólo hay un productor y un consumidor

## Múltiples productores y múltiples consumidores



Si hay más de un productor y/o más de un consumidor, las operaciones de depositar y retirar en sí mismas son SC y deben ejecutar con exclusión mutua

Si no se protege cada slot, podría retirarse 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
  }
}

```

The diagram shows a circular buffer structure labeled 'Cyclic Buffer'. The buffer is divided into 12 slots, numbered 0 through 11 around the perimeter. A black dot is placed in slot 0. An arrow labeled 'front' points to slot 0. An arrow labeled 'rear' points to slot 11. The label 'n-1' is placed near slot 11.

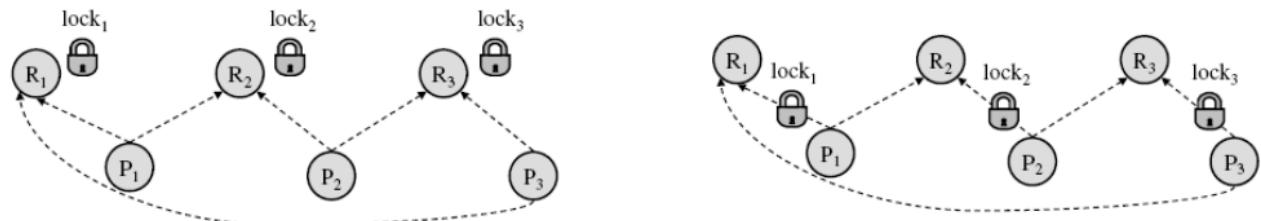
## Varios procesos compitiendo por varios recursos compartidos

Problema de varios procesos (P) y varios recursos (R) cada uno protegido por un lock

Un proceso debe adquirir los locks de todos los recursos que necesita

Puede caerse en deadlock cuando varios procesos compiten por conjuntos superpuestos de recursos

Por ejemplo: cada  $P[i]$  necesita  $R[i]$  y  $R[(i+1) \bmod n]$



## Problema de los filósofos: exclusión mutua selectiva

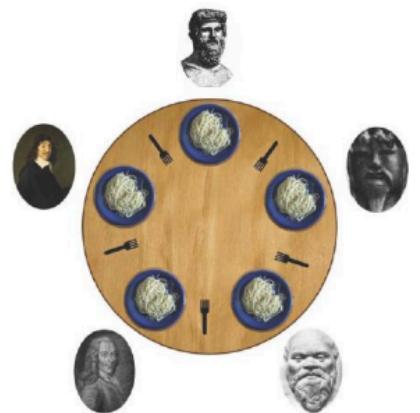
Problema de exclusión mutua entre procesos que compiten por el acceso a conjuntos superpuestos de variables compartidas

### **Problema de los filósofos:**

```

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

```



Cada tenedor es una SC: puede ser tomado por un único filósofo a la vez, entonces pueden representarse los tenedores por un arreglo de semáforos

Levantar un tenedor sería P

Bajar un tenedor sería V

Cada filósofo necesita el tenedor izquierdo y el derecho

Si todos los filósofos a la vez agarran primero el de la izquierda, cuando vayan a agarrar el de la derecha no van a poder => hay deadlock

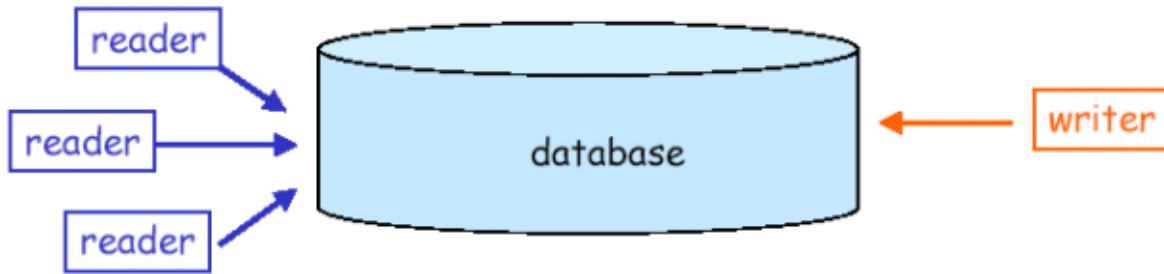
Solución sin deadlock: uno de los filósofos (el 4) agarra los tenedores en sentido inverso que el resto (primero la derecha y luego la izquierda). De esta manera, se elimina la espera cíclica

```
sem tenedores [5] = {1,1,1,1,1};  
  
process Filosofos[i = 0..3]  
{ while(true)  
  { P(tenedor[i]); P(tenedor[i+1]);  
    comer;  
    V(tenedor[i]); V(tenedor[i+1]);  
  }  
}  
  
process Filosofos[4]  
{ while(true)  
  { P(tenedor[0]); P(tenedor[4]);  
    comer;  
    V(tenedor[0]); V(tenedor[4]);  
  }  
}
```

## Lectores y escritores

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



Procesos asimétricos y, según el scheduler, con diferente prioridad

Es también un problema de exclusión mutua selectiva: clases de procesos compiten por el acceso a la BD

Diferentes soluciones:

- Como problema de exclusión mutua
- Como problema de sincronización por condición

Solución con exclusión mutua

Los escritores necesitan acceso mutuamente exclusivo

Los lectores (como grupo) necesitan acceso exclusivo con respecto a cualquier escritor

```
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);
  }
}
```

Esta solución le da preferencia a los lectores (no es fair)

Solución con sincronización por condición

```
int nr = 0, nw = 0;  
process Lector [i = 1 to M]           process Escritor [j = 1 to N]  
{ while(true)                      { while(true)  
{ ...                                { ...  
  < await (nw == 0) nr = nr + 1; >    < await (nr==0 and nw==0) nw=nw+1; >  
  lee la BD;                         escribe la BD;  
  < nr = nr - 1; >                  < nw = nw - 1; >  
 } } } }
```

En algunos casos, await puede ser implementada directamente usando semáforos u otras operaciones primitivas. Pero no siempre...

En el caso de las guardas de los await en la solución anterior, se superponen en que el protocolo de entrada para escritores necesita que tanto nw como nr sean 0, mientras para lectores sólo que nw sea 0. Ningún semáforo podría discriminar entre estas condiciones => passing the baton

Passing the baton

Técnica general para implementar sentencias await

Cuando un proceso está dentro de una SC mantiene el baton (testimonio, token) que significa permiso para ejecutar

Cuando el proceso llega a un SIGNAL (sale de la SC), pasa el baton (control) a otro proceso. Si ningún proceso está esperando por el baton (es decir esperando entrar a la SC) el baton se libera para que lo tome el próximo proceso que trata de entrar

La sincronización se expresa con sentencias atómicas de la forma:

$F_1 : \langle S_i \rangle$  o  $F_2 : \langle \text{await } (B_j) S_j \rangle$

Puede hacerse con semáforos binarios divididos (SBS).

$e$  semáforo binario inicialmente  $1$  (controla la entrada a sentencias atómicas).

Utilizamos un semáforo  $b_j$  y un contador  $d_j$  cada uno con guarda diferente  $B_j$ ; todos inicialmente  $0$ .

$b_j$  se usa para demorar procesos esperando que  $B_j$  sea *true*.

$d_j$  es un contador del número de procesos demorados sobre  $b_j$ .

$e$  y los  $b_j$  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;**

$\langle S_i \rangle$

$F_2 : P(e);$

**if (not  $B_j$ ) {** $d_j = d_j + 1; V(e); P(b_j); }$

$S_j;$

**SIGNAL**

$\langle \text{await } (B_j) S_j \rangle$

**SIGNAL:** **if ( $B_1$  and  $d_1 > 0$ ) {** $d_1 = d_1 - 1; V(b_1)$ **}**

**□ ...**

**□ ( $B_n$  and  $d_n > 0$ ) {** $d_n = d_n - 1; V(b_n)$ **}**

**□ else  $V(e);$**

**fi**

```

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 **SIGNAL<sub>i</sub>** es el de señalizar *exactamente* a uno de los semáforos  $\Rightarrow$  los procesos se van pasando el **batón**.

**SIGNAL<sub>i</sub>** es una abreviación de:

```

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

```

Algunos de los SIGNAL se pueden simplificar.

```

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)

Varios procesos que compiten por el uso de un recurso compartido de una sola unidad

request (tiempo,id). Si el recurso está libre, es alocado inmediatamente al proceso id; sino, el proceso id se demora

release ()

Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) con el mínimo valor de tiempo. Si dos o más procesos tienen el mismo valor de tiempo, el recurso es alocado al que esperó más

SJN minimiza el tiempo promedio de ejecución, aunque es unfair (¿por qué?)

Puede mejorarse con la técnica de aging (dando preferencia a un proceso que esperó mucho tiempo)

Para el caso general de alocación de recursos (no SJN):

bool libre = true;

**request (tiempo,id):** <await (libre) libre = false;>

**release ():** <libre = true;>

En SJN, un proceso que invoca a request debe demorarse hasta que el recurso esté libre y su pedido sea el próximo en ser atendido de acuerdo a la política. El parámetro tiempo entra en juego sólo si un pedido debe ser demorado

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

**request (tiempo, id):**

```
P(e);  
if (not libre) DELAY;  
libre = false;  
SIGNAL;
```

**release ( ):**

```
P(e);  
libre = true;  
SIGNAL;
```

En **DELAY** un proceso:

- Inserta sus parámetros en un conjunto, cola o lista de espera (pares)
- Libera la SC ejecutando V(e)
- Se demora en un semáforo hasta que request puede ser satisfecho

En **SIGNAL** un proceso:

- Cuando el recurso es liberado, si pares no está vacío, el recurso es asignado a un proceso de acuerdo a SJN

Cada proceso tiene una condición de demora distinta, dependiendo de su posición en pares

El proceso id se demora sobre el semáforo b[id]

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

**request(tiempo,id):** P(e);

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

**release( ):** P(e);

```
libre = true;  
if (Pares  $\neq \emptyset$  ) { remover el primer par (tiempo,id) de Pares; V(b[id]); }  
else V(e);
```

Una variable `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 privados

# Clase 5 - Monitores

## Problemas con los semáforos

Variables compartidas globales a los procesos

Sentencias de control de acceso a la sección crítica dispersas en el código

Al agregar procesos, se debe verificar el acceso correcto a las variables compartidas

Aunque exclusión mutua y sincronización por condición son conceptos distintos, se programan de forma similar

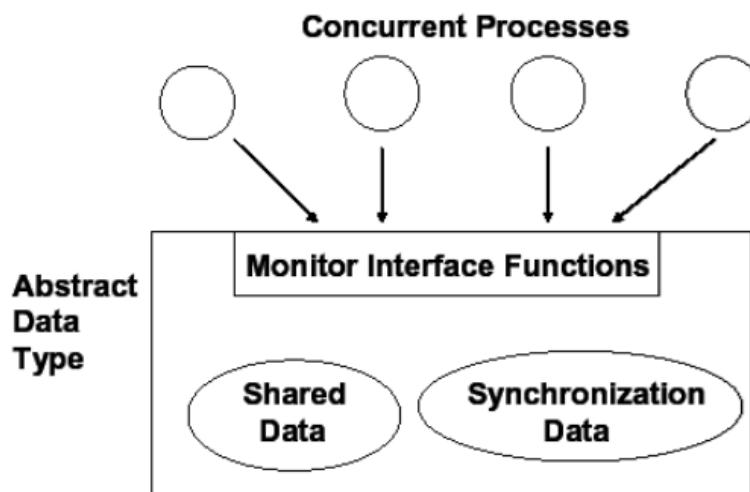
## Monitores

Módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos

Mecanismo de abstracción de datos:

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

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



## Exclusión mutua

Implícita asegurando que los procedures en el mismo monitor no se ejecutan concurrentemente

## Sincronización por condición

Explícita con variables condición

## Programa concurrente

Va a estar formado por procesos activos y monitores pasivos

Dos procesos interactúan invocando procedures de un monitor

Ventajas:

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

## Notación

Un monitor agrupa la representación y la implementación de un recurso compartido

Se distingue a un monitor de un TAD en procesos secuenciales en que es compartido por procesos que se ejecutan concurrentemente

Tiene interfaz y cuerpo:

- La interfaz especifica operaciones que brinda el recurso
- El cuerpo tiene variables que representan el estado del recurso y procedures que implementan las operaciones de la interfaz

Sólo los nombres de los procedures son visibles desde afuera

Sintácticamente, los llamados al monitor tienen la forma:

NombreMonitor.op<sub>i</sub>(argumentos)

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

El programador de un monitor no puede conocer a priori el orden de llamado

```
monitor NombreMonitor {  
    declaraciones de variables permanentes;  
    código de inicialización
```

```
procedure op1 (par. formales1)
```

```
{  cuerpo de op1  
}
```

```
.....
```

```
procedure opn (par. formalesn)
```

```
{  cuerpo de opn  
}
```

```
}
```

## Sincronización

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

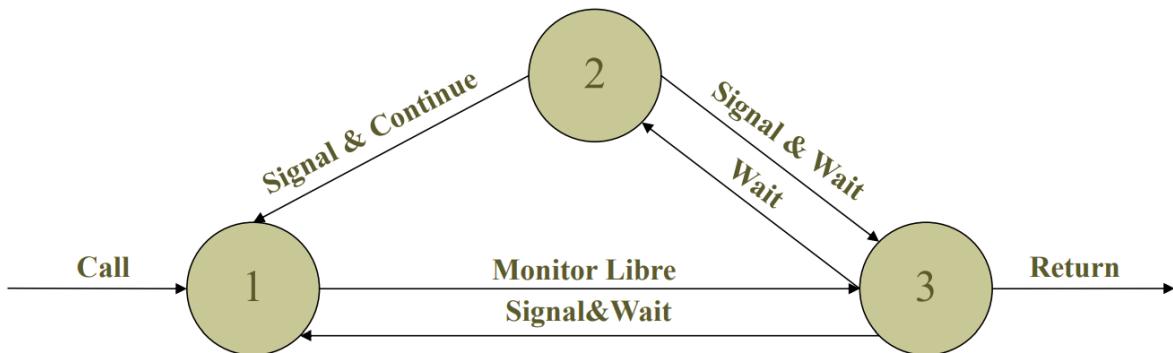
El valor asociado a cv es una cola de procesos demorados, no visible directamente al programador

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 se podrá ejecutar cuando readquiera el acceso exclusivo al monitor
- signal\_all(cv) → despierta todos los procesos demorados en cv, quedando vacía la cola asociada a cv

Disciplinas de señalización:

- Signal and continue (es el utilizado en la materia): 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 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)



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

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

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

`wait(cv, rank)` → el proceso se demora en la cola de cv en orden ascendente de acuerdo al parámetro rank y deja el acceso exclusivo al monitor

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

## Wait/signal vs P/V

<b>WAIT</b>	<b>P</b>
El proceso siempre se duerme	El proceso sólo se duerme si el semáforo es 0.

<b>SIGNAL</b>	<b>V</b>
Si hay procesos dormidos despierta al primero de ellos. En caso contrario no tiene efecto posterior.	Incrementa el semáforo para que un proceso dormido o que hará un P continue. No sigue ningún orden al despertarlos.

# Clase 6 - Memoria distribuida y pasaje de mensajes asincrónico

Arquitecturas de memoria distribuida => procesadores + memoria local + red de comunicaciones + mecanismo de comunicación / sincronización => intercambio de mensajes

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

Van a ser necesarias primitivas de pasaje de mensajes: se va a necesitar una interfaz con el sistema de comunicaciones => semáforos + datos + sincronización

Lo único que van a compartir los procesos son canales (físicos o lógicos)

Variantes para los canales:

- Cuántos procesos pueden enviar o recibir datos por esos canales:
  - Mailbox: canales totalmente compartidos, conocidos por todos los procesos
  - Input port: un único proceso puede ser receptor, pero cualquier proceso podrá ser emisor
  - Link: es un canal punto a punto, un único proceso puede enviar y un único proceso puede recibir
- Uni o bidireccionales
- Sincrónicos (ambos procesos a comunicarse deben esperar a llegar al punto de comunicación para poder realizar dicha comunicación, y una vez terminada recién podrán continuar con su trabajo) o asincrónicos

## Características

Los canales son lo único que comparten los procesos:

- Variables locales a un proceso (“cuidador”)
- La exclusión mutua no requiere mecanismo especial
- Los procesos interactúan comunicándose
- Accedidos por primitivas de envío y recepción

## Mecanismos para el Procesamiento Distribuido:

- Pasaje de Mensajes Asincrónicos (PMA) (canales de tipo mailbox, unidireccionales, asincrónicos)
- Pasaje de Mensajes Sincrónico (PMS) (canales de tipo link, unidireccionales, sincrónicos)
- Llamado a Procedimientos Remotos (RPC) (canales de tipo link, bidireccionales, sincrónicos)
- Rendezvous (canales de tipo link, bidireccionales, sincrónicos)

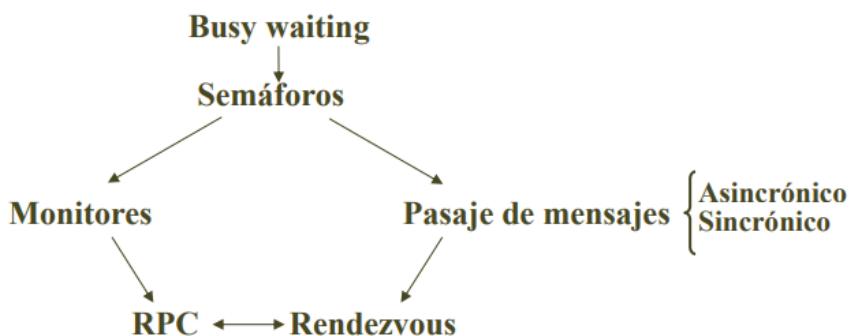
## Patrones de interacción:

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

Cada mecanismo es más adecuado para determinados patrones

## Relación entre mecanismos de sincronización

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



## Pasaje de mensajes asincrónico (PMA)

En PMA los canales son colas de mensajes enviados y aún no recibidos

Declaración de canales: chan <nombre\_canal>(id<sub>1</sub>: tipo<sub>1</sub> ,..., id<sub>n</sub> : tipo<sub>n</sub>)

Ejemplos de declaración de canales:

- chan entrada (char);
- chan acceso\_disco (INT cilindro, INT bloque, INT cant, CHAR\* buffer);
- chan resultado[n] (INT);

Operación Send (atómica): un proceso agrega un mensaje al final de la cola (“ilimitada”) de un canal ejecutando un send, que no bloquea al emisor: send <nombre\_canal>(expr<sub>1</sub> ,..., expr<sub>n</sub>);

Operación Receive (atómica): 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 <nombre\_canal>(var<sub>1</sub> ,..., var<sub>n</sub>);

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

Receive es una primitiva bloqueante, ya que produce un delay

Semántica: el proceso NO hace nada hasta recibir un mensaje en la cola correspondiente al canal. NO es necesario hacer polling

## Características de los canales

Acceso a los contenidos de cada canal: atómico y respeta orden FIFO

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

Se supone que los mensajes no se pierden ni modifican y que todo mensaje enviado en algún momento puede ser “leído”

empty(ch): determina si la cola de un canal está vacía

Útil cuando el proceso puede hacer trabajo productivo mientras espera un mensaje, pero debe usarse con cuidado:

- O podría ser false, y no haber más mensajes cuando sigue ejecutando (si no es el único en recibir por ese canal)
- O la evaluación de empty podría ser true, y sin embargo 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. Según la forma en que se usan podría ser:

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

## Productores y consumidores (filtro)

Filtro: 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

Esta función del filtro puede especificarse por un predicado que relacione los valores de los mensajes de salida con los de entrada

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)

### Process Sort

```
{ receive todos los números del canal entrada;  
  ordenar los números;  
  send de los números ordenados por el canal OUTPUT;  
 }
```

¿Cómo determina Sort que recibió todos los números?

- Conoce  $N$
- Envía  $N$  como el primer elemento a recibir por el canal entrada
- Cierra la lista de  $N$  números con un valor especial o “centinela”

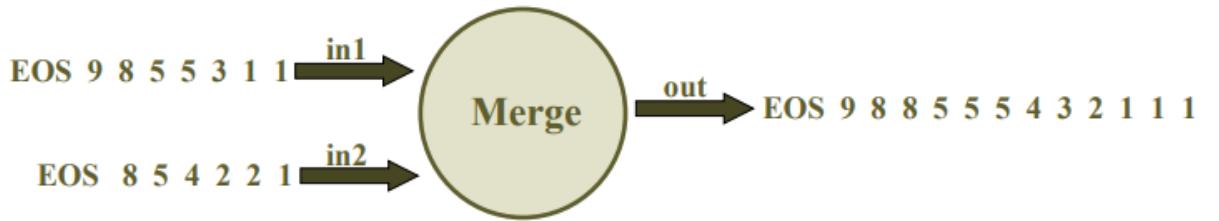
Solución más eficiente que la “secuencial” => red de pequeños procesos que ejecutan en paralelo e interactúan para “armar” la salida ordenada (merge network)

Idea: mezclar repetidamente y en paralelo dos listas ordenadas de  $N_1$  elementos cada una en una lista ordenada de  $2 \cdot N_1$  elementos

Con PMA, pensamos en 2 canales de entrada por cada canal de salida, y un carácter especial EOS cerrará cada lista parcial ordenada

La red es construida con filtros Merge:

- Cada Merge recibe valores de dos streams de entrada ordenados, in1 e in2, y produce un stream de salida ordenado, out
- Los streams terminan en EOS, y Merge agrega EOS al final
- ¿Cómo implemento Merge?. Comparar repetidamente los próximos dos valores recibidos desde in1 e in2 y enviar el menor a out



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

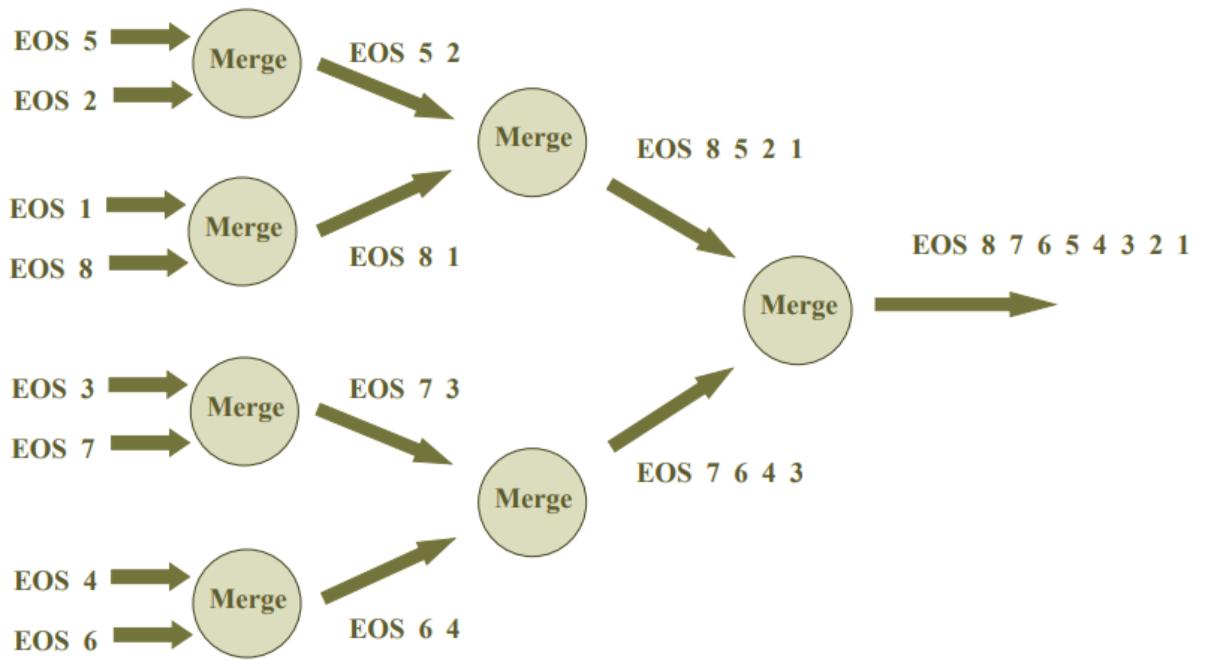
*Process Merge*

```
{ 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);
}
```



$n-1$  procesos; el ancho de la red es  $\log_2 n$

Canales de entrada y salida compartidos

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)

Los filtros podemos conectarlos de distintas maneras. Solo se necesita que la salida de uno cumpla las suposiciones de entrada del otro => pueden reemplazarse si se mantienen los comportamientos de entrada y salida

## Cliente servidor

Servidor: proceso que maneja pedidos (“requests”) de otros procesos clientes

¿Cómo implementamos C/S con PMA?

Dualidad entre monitores y PM: cada uno de ellos puede simular al otro

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

```
monitor Mname
{ declaración de variables permanentes;
  código de inicialización;
  procedure op(formales) { cuerpo de op; }
}
```

Un proceso Cliente que envía un mensaje a un canal de requerimientos general, luego recibe el resultado desde un canal de respuesta propio (los canales siguen siendo mailbox, pero se van a comportar como input port y link respectivamente)

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 ( $N \leq M$ ).

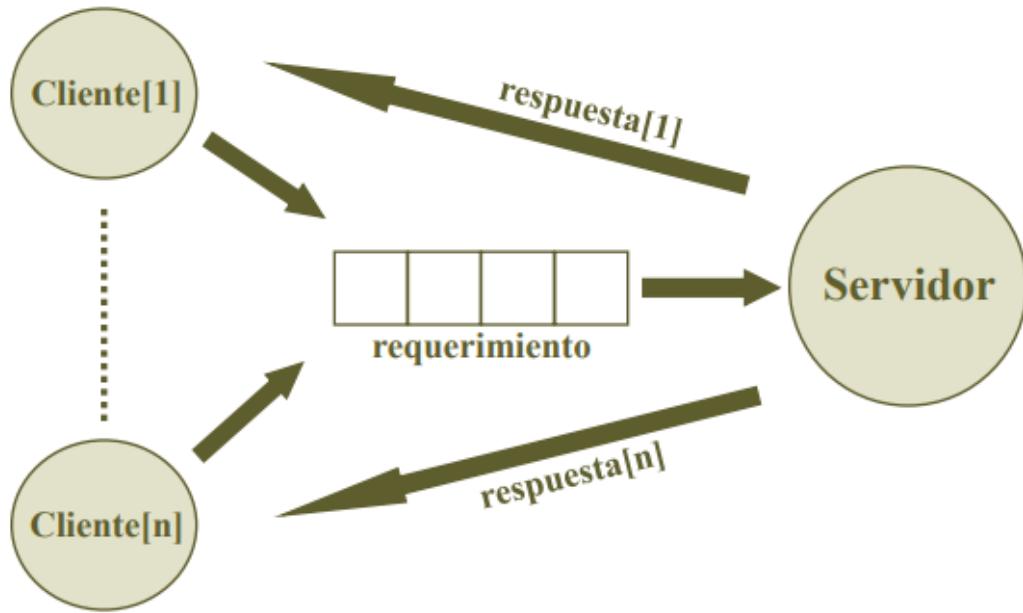
1 operación

Para simular al monitor *Mname*, usamos un proceso server Servidor

Las variables permanentes serán variables locales de Servidor

Llamado: un proceso cliente envía un mensaje a un canal de requerimiento

Luego recibe el resultado por un canal de respuesta propio



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);
}
  
```

Múltiples operaciones

Podemos generalizar esta solución de C/S con una única operación para considerar múltiples operaciones

El IF del Servidor será un CASE con las distintas clases de operaciones

El cuerpo de cada operación toma datos de un canal de entrada en args y los devuelve al cliente adecuado en resultados

```
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; }

      .....
      elseif ( 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);
}
```

## Múltiples operaciones y variables de condición

Hasta ahora el monitor no requería variables condición ya que el Servidor no requería demorar la atención de un pedido de servicio

Caso general: monitor con múltiples operaciones y con sincronización por condición

Para los clientes, la situación es transparente => cambia el servidor

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

- Los clientes “adquieren” y devuelven unidades del recurso
- Las unidades libres se insertan en un “conjunto” sobre el que se harán las operaciones de INSERTAR y REMOVER
- El número de unidades disponibles es lo que “controla” nuestra variable de sincronización por condición

```
Monitor Administrador Recurso
{ int disponible = MAXUNIDADES;
  set unidades = valores iniciales;
  cond libre;

  procedure adquirir( int *Id )
  { if (disponible == 0) wait(libre)
    else disponible--;
    remove(unidades, id);
  }

  procedure liberar(int id )
  { insert(unidades, id);
    if (empty(libre)) disponible ++
    else signal(libre);
  }
}
```

Caso en que el servidor tiene dos operaciones:

- Si no hay unidades disponibles, el servidor no puede esperar hasta responder al pedido debe salvarlo y diferir la respuesta
- Cuando una unidad es liberada, atiende un pedido salvado (si hay) enviando la unidad

```

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
  }
}

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);
}

```

El monitor y el servidor muestran la dualidad entre monitores y PM: hay una correspondencia directa entre los mecanismos de ambos

La eficiencia de monitores o PM depende de la arquitectura física de soporte:

- Con MC conviene la invocación a procedimientos y la operación sobre variables condición
- Con arquitecturas físicamente distribuidas tienden a ser más eficientes los mecanismos de PM

Dualidad entre Monitores y Pasaje de Mensajes

### **Programas con Monitores**

- Variables permanentes
- Identificadores de procedures
- Llamado a procedure
- Entry del monitor
- Retorno del procedure
- Sentencia *wait*
- Sentencia *signal*
- Cuerpos de los procedure

### **Programas basados en PM**

- |                                 |   |
|---------------------------------|---|
| ↔ Variables permanentes         | ↔ Variables locales del servidor                            |
| ↔ Identificadores de procedures | ↔ Canal <i>request</i> y tipos de operación                 |
| ↔ Llamado a procedure           | ↔ <i>send request()</i> ; <i>receive respuesta</i>          |
| ↔ Entry del monitor             | ↔ <i>receive request()</i>                                  |
| ↔ Retorno del procedure         | ↔ <i>send respuesta()</i>                                   |
| ↔ Sentencia <i>wait</i>         | ↔ Salvar pedido pendiente                                   |
| ↔ Sentencia <i>signal</i>       | ↔ Recuperar/ procesar pedido pendiente                      |
| ↔ Cuerpos de los procedure      | ↔ Sentencias del “case” de acuerdo a la clase de operación. |

## Sentencias de alternativa múltiples

```
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);
}
```

Usando sentencias de alternativa múltiples se hace uso de un canal por cada tipo de operación disponible

De esta manera, el servidor puede chequear (usando empty(chan)) si hay pedidos de algún tipo de operación y de esa manera, va a tratar no deterministicamente a uno de los pedidos que pueda llevar a cabo (si no hay unidades disponibles, no va a tratar las solicitudes de adquirir una unidad)

Usando esta opción, se evitan tener que hacer colas de pedidos pendientes

Si ninguna de las condiciones se cumple, al no haber ningún mensaje para recibir, se generaría busy waiting. En memoria distribuida no es un problema porque cada proceso está corriendo en una unidad de procesamiento diferente. Si bien no es aconsejable, va a ser permitido cuando cualquier otro tipo de solución sea demasiado compleja

## Continuidad conversacional

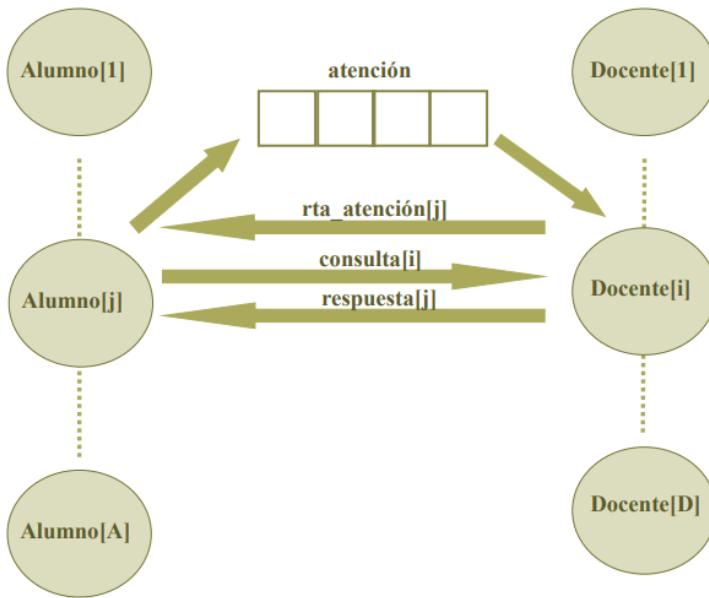
Existen A alumnos que hacen consultas a D docentes

El alumno espera hasta que un docente lo atiende, y a partir de ahí le comienza a realizar las diferentes consultas hasta que no le queden dudas

Los alumnos son los procesos “clientes”, y los docentes los procesos “servidores”

Los procesos servidores son idénticos, y cualquiera de ellos que esté libre puede atender un requerimiento de un alumno

Todos los alumnos pueden pedir atención por un canal global y recibirán respuesta de un docente dado por un canal propio



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

Este ejemplo de interacción entre clientes y servidores se denomina continuidad conversacional (desde la solicitud de atención hasta la última consulta)

atención es un canal compartido por el que cualquier docente puede recibir

Si cada canal puede tener un solo receptor, se necesita otro proceso intermedio.

¿Para qué?

## 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

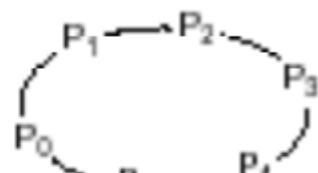
Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: centralizado, simétrico y en anillo circular



(a) Centralized solution



(b) Symmetric solution



(c) Ring solution

### Solución centralizada

Cada proceso tiene un valor  $V$  local. Al final todos los procesos deben conocer el mínimo y máximo valor de todo el sistema

La arquitectura centralizada es apta para una solución en que todos envían su dato local  $V$  al procesador central, éste ordena los  $N$  datos y reenvía la información del mayor y menor a todos los procesos  $\Rightarrow 2(N-1)$  mensajes

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

```
Process P[0]
{ int v; int nuevo, minimo = v, maximo = 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, maximo;
  send valores (v);
  receive resultados[i-1](minimo, maximo);
}
```

## 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 trasmite su dato local  $V$  a los  $N-1$  restantes procesos

Luego recibe y procesa los  $N-1$  datos que le faltan, de modo que en paralelo toda la arquitectura está calculando el mínimo y el máximo y toda la arquitectura tiene acceso a los  $N$  datos

Ejemplo de solución SPMD (Single Program Multi-Data): cada proceso ejecuta el mismo programa pero trabaja sobre datos distintos =>  $N(N-1)$  mensajes

Si disponemos de una primitiva de broadcast, serán nuevamente  $N$  mensajes

```
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 > máximo) máximo = nuevo;
    }
}
```

## Solución anillo circular

Un tercer modo de organizar la solución es tener un anillo donde  $P[i]$  recibe mensajes de  $P[i-1]$  y envía mensajes a  $P[i+1]$ .  $P[n-1]$  tiene como sucesor a  $P[0]$

Esquema de 2 etapas:

- En la primera cada proceso recibe dos valores y los compara con su valor local, trasmitiendo un máximo local y un mínimo local a su sucesor
- En la segunda etapa todos deben recibir la circulación del máximo y el mínimo global

$P[0]$  deberá ser algo diferente para “arrancar” el procesamiento

Se requerirán  $(2N)-1$  mensajes

Notar que si bien el número de mensajes es lineal (igual que en la centralizada) los tiempos pueden ser muy diferentes

```

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);
};

```

## Análisis sobre las soluciones

Simétrica es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast)

Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup

Centralizada y anillo usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance:

- En centralizada, los mensajes al coordinador se envían casi al mismo tiempo => sólo el primer receive del coordinador demora mucho
- En anillo, todos los procesos son productores y consumidores. El último tiene que esperar a que todos los otros (uno por vez) reciban un mensaje, hacer poco cómputo, y enviar su resultado. Los mensajes circulan 2 veces completas por el anillo Solución inherentemente lineal y lenta para este problema

# Clase 7 - Pasaje de mensajes sincrónico y paradigmas de interacción entre procesos

Los canales son de tipo link o punto a punto (1 emisor y 1 receptor), unidireccionales y sincrónicos

La diferencia entre PMA y PMS es la primitiva de transmisión Send. En PMS es bloqueante y la llamaremos (por ahora) sync\_send:

- El trasmisor queda esperando que el mensaje sea recibido por el receptor
- La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje => menos memoria
- Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (los emisores se bloquean)

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

## 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  
  }  
}
```



Si el productor y el consumidor tardan lo mismo en resolver sus cálculos, el tiempo de ejecución es  $N + \langle\text{tiempo\_para\_una\_unidad}\rangle$ , ya que mientras el productor produce, el consumidor consume y se encuentran al mismo tiempo en el send/receive (excepto en la primera producción). Para este caso es igual PMS a PMA (en cuanto a tiempo de ejecución)

Si el productor es más rápido que el consumidor, el tiempo de ejecución es relativo al consumidor, ya que mientras el consumidor consume, el productor, que ya va a haber terminado de producir, se quedará bloqueado esperando a que el consumidor reciba su producción. Para este caso es igual PMS a PMA (en cuanto a tiempo de ejecución). Esto es porque, tanto en PMS como en PMA, el tiempo final está determinado por el consumidor, solo que en PMA, el productor, podrá terminar su ejecución antes que en PMS

Si el consumidor es más rápido que el productor, el tiempo de ejecución es relativo al productor, ya que mientras el productor produce, el consumidor, que ya va a haber terminado de consumir, se quedará bloqueado esperando a que el productor le envíe su producción. Para este caso es igual PMS a PMA (en cuanto a tiempo de ejecución). Esto es porque, tanto en PMS como en PMA, el tiempo final está determinado por el productor. En este caso, PMA y PMS son iguales en su ejecución, ya que, para ambos casos, el consumidor debe esperar a que el productor produzca para poder consumir lo producido

Si los cálculos del productor se realizan mucho más rápido que los del consumidor en las primeras  $n/2$  operaciones (por ejemplo en 1 segundo el productor y 10 segundos el consumidor), y luego se realizan mucho más lento durante otras  $n/2$  interacciones (por ejemplo en 1 segundo el consumidor y 10 segundos el productor):

- 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. Se tardará en el ejemplo,  $10*N$  segundos en completar la ejecución
- 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. Se tardará en el ejemplo,  $N/2 + 10*N/2 = 11*N/2$  segundos en completar la ejecución, ya que  $N/2$  es lo que tarda el productor en producir los primeros  $N/2$  elementos, y luego tarda  $10*N/2$  en producir los últimos  $N/2$  elementos

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

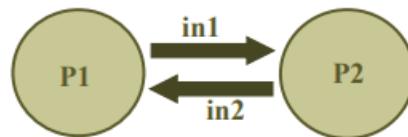
¿Qué pasa si existe más de un productor/consumidor?

La concurrencia también se reduce en algunas interacciones Cliente/Servidor:

- Cuando un cliente está liberando un recurso, no habría motivos para demorarlo hasta que el servidor reciba el mensaje, pero con PMS se tiene que demorar
- Otro ejemplo se da cuando un cliente quiere escribir en un display gráfico, un archivo u otro dispositivo manejado por un proceso servidor. Normalmente el cliente quiere seguir inmediatamente después de un pedido de write

Otra desventaja del PMS es la mayor probabilidad de deadlock. El programador debe ser cuidadoso de que todas las sentencias send (sync\_send) y receive hagan matching

## Deadlock en PMS



chan in1(int), in2(int);

Process P1  
{ int valorA = 1, valorB;  
sync\_send in2(valorA);  
receive in1(valorB);  
}

Process P2  
{ int valorA, valorB = 2;  
sync\_send in1(valorB);  
receive in2 (valorA);  
}

chan in1(int), in2(int);

Process P1  
{ int valorA = 1, valorB;  
sync\_send in2(valorA);  
receive in1(valorB);  
}

Process P2  
{ int valorA, valorB = 2;  
receive in2 (valorA);  
sync\_send in1(valorB);  
}

## CSP

CSP (Communicating Sequential Processes, Hoare 1978) fue uno de los desarrollos fundamentales en Programación Concurrente. Muchos lenguajes reales (OCCAM, ADA, SR) se basan en CSP

Las ideas básicas introducidas por Hoare fueron PMS y comunicación guardada: PM con waiting selectivo

Canal: link directo entre dos procesos en lugar de mailbox global. Son half duplex (unidireccionales) y nominados (no deben ser declarados, solo se nombra el proceso con el que se quiere comunicar)

Las sentencias de Entrada (?) o query y Salida (! o shriek o bang) son el único medio por el cual los procesos se comunican. process A { . . . B ! e; . . . } process B { . . . A ? x; . . . }

Para que se produzca la comunicación, deben matchear, y luego se ejecutan simultáneamente

Efecto: sentencia de asignación distribuida

Formas generales de las sentencias de comunicación:

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

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

Destino y Fuente nombran un proceso simple, o un elemento de un arreglo de procesos

Fuente puede nombrar cualquier elemento de un arreglo (Fuente[\*]), solo sirve en la recepción

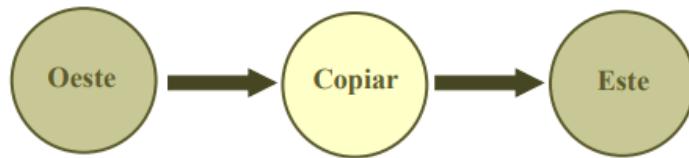
port son etiquetas que se usan para distinguir entre distintas clases de mensajes que un proceso podría recibir (puede omitirse si es sólo uno)

Dos procesos se comunican cuando ejecutan sentencias de comunicación que hacen matching. A ! canaluno(dato); B ? canaluno(resultado);

## Ejemplos básicos

- *Proceso filtro que copia caracteres recibidos del proceso Oeste al proceso Este:*

Process Oeste { char c; do true → <i>Generar(c);</i> <i>Copiar ! (c);</i> od }	Process Copiar { char c; do true → <i>Oeste ? (c) ;</i> <i>Este ! (c);</i> od }	Process Este { char c; do true → <i>Copiar ? (c) ;</i> <i>Usar(c);</i> od }
--	---	---



Server que calcula el MCD de dos enteros con el algoritmo de Euclides. **MCD** espera recibir entrada en su port **args** desde un cliente. Envía la respuesta al port **resultado** del cliente.

```
Process MCD
{ int id, x, y;
do true →
    Cliente[*] ? args(id, x, y);
    do x > y → x := x - y;
    □ x < y → y := y - x;
    od
    Cliente[id] ! resultado(x);
od
}
```

- *Cliente[i]* se comunica con *MCD* ejecutando:

```
...
MCD ! args(i, v1, v2);
MCD ? resultado(r);
...
```

## Comunicación guardada

Limitaciones de ? y ! ya que son bloqueantes

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

Por ejemplo, el proceso Copiar podría extenderse para hacer buffering de k caracteres: si hay más de 1 pero menos de k caracteres en el buffer, Copiar podría recibir otro carácter o sacar 1

Las operaciones de comunicación (? y !) pueden ser guardadas, es decir hacer un AWAIT hasta que una condición sea verdadera

El do e if de CSP usan los comandos guardados de Dijkstra ( $B \rightarrow S$ )

Las sentencias de comunicación guardada soportan comunicación no determinística:

$B; C \rightarrow S;$

- B es una condición booleana (puede omitirse y si eso sucede, se asume true)
- C es una sentencia de comunicación envío o recepción (en la práctica solo sera recepción)
- S es el conjunto de secuencias que se van a ejecutar en caso de que haya éxito
- B y C forman la guarda
- La guarda tiene éxito si B es true y ejecutar C no causa demora (en el caso de una recepción implicaría que el proceso que tiene que enviar el dato, ya realizó dicho envío)
- La guarda falla si B es falsa
- La guarda se bloquea si B es true pero C no puede ejecutarse inmediatamente

Las sentencias de comunicación guardadas aparecen en if y do:

```
if      B1 ; comunicación1 → S1 ;  
      B2 ; comunicación2 → S2 ;  
fi
```

Ejecución:

1. 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 deterministicamente)
- Si algunas guardas se bloquean (y no hay ninguna exitosa), se espera hasta que alguna de ellas tenga éxito

2. Luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación de la guarda elegida
3. Se ejecuta la sentencia Si

La ejecución del do es similar, pero se repite hasta que todas las guardas fallen

## Ejemplos

Podemos re-programar Copiar para usar comunicación guardada:

```
Process Copiar
  { char c;
    do Oeste ? (c) → Este!(c);
    od
  }
```

Extendemos *Copiar* para manejar un buffer de tamaño 2. Luego de ingresar un carácter, el proceso que copia puede estar recibiendo un segundo carácter de Oeste o enviando uno a Este.

```
Process Copiar2
  { char c1, c2;
    Oeste ? (c1);
    do Oeste ? (c2) → Este ! (c1) ;
      c1=c2;
    □ Este ! (c1) → Oeste? (c1);
    od
  }
```

Copiar con buffer limitado

### Copiar con un buffer limitado

```
Process Copiar
{ char buffer[80];
  int front = 0, rear = 0, cantidad = 0;
  do cantidad <80; Oeste?(buffer[rear]) → cantidad = cantidad + 1;
                                         rear = (rear + 1) MOD 80;
  □ cantidad >0; Este!(buffer[front]) → cantidad := cantidad - 1;
                                         front := (front + 1) MOD 80;
  od
}
```

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

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

Asignación de recursos

### Asignación de Recursos

```
Process Alocador
{ int disponible = MaxUnidades;
  set unidades = valores iniciales;
  int indice, idUnidad;

  do disponible >0; cliente[*] ? acquire(indice) → disponible = disponible -1;
                                         remove (unidades, idUnidad);
                                         cliente[indice] ! reply(idUnidad);
  □ cliente[*] ? release(indice, idUnidad) → disponible = disponible +1;
                                         insert (unidades, idUnidad);
  od
}
```

La solución es concisa. Usa múltiples **ports** y un brazo del **do** para atender cada una. Se demora en un mensaje **acquire** hasta que haya unidades, y no es necesario salvar los pedidos pendientes.

## Intercambio de valores

### Intercambio de Valores

```
Process P1
{  int valor1 = 1, valor2;
   if P2 ! (valor1) → P2 ? (valor2);
   □ P2 ? (valor2) → P2 ! (valor1);
   fi
}
```

```
Process P2
{  int valor1 , valor2 = 2;
   if P1 ! (valor2) → P1 ? (valor1);
   □ P1 ? (valor1) → P1 ! (valor2);
   fi
}
```

Esta solución simétrica **NO** tiene *deadlock* porque el no determinismo en ambos procesos hace que se acoplen las comunicaciones correctamente. Si bien es simétrica, es más compleja que la de PMA...

## La Criba de Eratóstenes para generar números primos

Problema: generar todos los primos entre 2 y n → 2 3 4 5 6 7 8 . . . N

Comenzando con el primer número (2), recorremos la lista y borramos los múltiplos de ese número

Si n es impar: 2 3 5 7 . . . n

Pasamos al próximo número (3) y borramos sus múltiplos

Siguiendo hasta que todo número fue considerado, los que quedan son todos los primos entre 2 y n

La criba captura primos y deja caer múltiplos de los primos

¿Cómo paralelizar? Pipe de procesos filtro: cada uno recibe un stream de números de su predecesor y envía un stream a su sucesor. El primer número que recibe es el próximo primo, y pasa los no múltiplos

```

Process Criba[1]
{   int p = 2;
    for [i = 3 to n by 2] Criba[2] ! (i);
}

Process Criba[i = 2 to L]
{   int p, proximo;

    Criba[i-1] ? (p);
    do Criba[i-1] ? (proximo) →
        if ((proximo MOD p) <> 0 ) and (i < L) → Criba[i+1] ! (proximo);
    od
}

```

- El número total de procesos *Criba* ( $L$ ) debe ser lo suficientemente grande para garantizar que se generan todos los primos hasta  $n$ .
- Excepto *Criba[1]*, los procesos terminan bloqueados esperando un mensaje de su predecesor. Cuando el programa para, los valores de  $p$  en los procesos son los primos. Puede modificarse con centinelas.

### Ordenación de un arreglo

Problema: ordenar un arreglo de  $n$  valores en paralelo ( $n$  par, orden no decreciente)

Dos procesos  $P_1$  y  $P_2$ , cada uno inicialmente con  $n/2$  valores (arreglos  $a_1$  y  $a_2$  respectivamente)

Los  $n/2$  valores de cada proceso se encuentran ordenados inicialmente

Idea: realizar una serie de intercambios. En cada uno  $P_1$  y  $P_2$  intercambian  $a_1[\text{mayor}]$  y  $a_2[\text{menor}]$ , hasta que  $a_1[\text{mayor}] < a_2[\text{menor}]$

## Ordenación de un Arreglo

```
Process P1
{ int nuevo, a1[1:n/2]; const mayor = n/2;
  ordenar a1 en orden no decreciente
  P2 ! (a1[mayor]);
  P2 ? (nuevo);
  do a1[mayor] > nuevo →
    poner nuevo en el lugar correcto en a1, descartando el viejo a1[mayor]
    P2 ! (a1[mayor]);
    P2 ? (nuevo);
  od
}

Process P2
{ int nuevo, a2[1:n/2]; const menor = 1;
  ordenar a2 en orden no decreciente
  P1 ? (nuevo);
  P1 ! (a2[menor]);
  do a2[menor] < nuevo →
    poner nuevo en el lugar correcto en a2, descartando el viejo a2[menor]
    P1 ? (nuevo);
    P1 ? (a2[menor]);
  od
}
```

Notar que en la implementación del intercambio, las sentencias de entrada y salida son bloqueantes → usamos una solución asimétrica

Para evitar deadlock, P1 primero ejecuta una salida y luego una entrada, y P2 ejecuta primero una entrada y luego una salida

Comunicación guardada para programar una solución simétrica: esta solución es más costosa de implementar:

P1: ... if P2 ? (nuevo) → P2 ! (a1[mayor])  
    □ P2 ! (a1[mayor]) → P2 ? (nuevo)  
    fi...

P2: ... if P1 ? (nuevo) → P1 ! (a2[menor])  
    □ P1 ! (a2[menor]) → P1 ? (nuevo)  
    fi ...

Mejor caso → los procesos intercambian solo un par de valores

Peor caso → intercambian  $n/2 + 1$  valores:  $n/2$  para tener cada valor en el proceso correcto y uno para detectar terminación

Solución con  $b$  procesos  $P[1:b]$ , inicialmente con  $n/b$  valores cada uno  
 Cada uno primero ordena sus  $n/b$  valores  
 Luego ordenamos los  $n$  elementos usando aplicaciones paralelas repetidas  
 del algoritmo compare-and-exchange

Cada proceso ejecuta una serie de rondas:

- En las impares, cada proceso con número impar juega el rol de  $P_1$ , y cada proceso con número par el de  $P_2$
- En las rondas pares, cada proceso numerado par juega el rol de  $P_1$ , y cada proceso impar el rol de  $P_2$

### Ejemplo: $b=4$ - Algoritmo odd/even exchange sort

ronda	P[1]	P[2]	P[3]	P[4]
0	8	7	6	5
1	7	8	5	6
2	7	5	8	6
3	5	7	6	8
4	5	6	7	8

Cada intercambio progresiva hacia una lista totalmente ordenada  
 ¿Cómo pueden detectar los procesos si toda la lista está ordenada?  
 Un proceso individual no puede detectar que la lista entera está ordenada  
 después de una ronda pues conoce sólo dos porciones:

- Se puede usar un coordinador separado. Después de cada ronda, los procesos le dicen a éste si hicieron algún cambio a su porción =>  $2b$  mensajes de overhead en cada ronda
- Que cada proceso ejecute suficientes rondas para garantizar que la lista estará ordenada (en general, al menos  $b$  rondas)
  - Cada proceso intercambia hasta  $n/b+1$  mensajes por ronda
  - El algoritmo requiere hasta  $b \cdot 2^{*(n/b + 1)}$  intercambio de mensajes

# Paradigmas de interacción entre procesos

3 esquemas básicos de interacción entre procesos:

- Productor/consumidor
- Cliente/servidor
- Interacción entre pares

Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a otros paradigmas o modelos de interacción entre procesos:

- Paradigma 1: master / worker:
  - Implementación distribuida del modelo Bag of Task
- Paradigma 2: algoritmos heartbeat:
  - Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive
- Paradigma 3: algoritmos pipeline:
  - La información recorre una serie de procesos utilizando alguna forma de receive/send
- Paradigma 4: probes (send) y echoes(receive):
  - La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información
- Paradigma 5: algoritmos broadcast:
  - Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas
- Paradigma 6: token passing:
  - En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas
- Paradigma 7: servidores replicados:
  - Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos

## Manager/worker

El concepto de bag of tasks usando variables compartidas supone que un conjunto de workers comparten una “bolsa” con tareas independientes. Los workers sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa (ejemplo en LINDA manejando un espacio compartido de tuplas)

La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los workers

Analizaremos la implementación de este paradigma con mensajes en lugar de MC

Para esto un proceso manager implementará la “bolsa” manejando las tasks, comunicándose con los workers y detectando fin de tareas. Se trata de un esquema C/S (donde el servidor administra las tareas y los clientes hacen el trabajo)

Ejemplo: multiplicación de matrices ralas

## Heartbeat

Útil para soluciones iterativas que se quieren paralelizar

Usando un esquema “divide & conquer” se distribuye la carga (datos) entre los workers; cada uno es responsable de actualizar una parte

Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos

Cada “paso” 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;
      }
  }
```

En este caso, convendría usar comunicación asincrónica, ya que con comunicación sincrónica, se podría producir deadlock, y si se hace una solución donde eso no suceda, la misma no sería eficiente

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

## Topología de una red

Los procesadores están conectados por canales bidireccionales. Cada uno se comunica sólo con sus vecinos y conoce esos links. ¿Cómo puede cada procesador determinar la topología completa de la red?

Modelización:

- Procesador => proceso
- Links de comunicación => canales compartidos

Soluciones: los vecinos interactúan para intercambiar información local

Algoritmo Heartbeat: se expande enviando información; luego se contrae incorporando nueva información

Procesos  $\text{Nodo}[p:1..n]$

Vecinos de  $p$ :  $\text{vecinos}[1:n] \rightarrow \text{vecinos}[q]$  es true si  $q$  es vecino de  $p$

Problema: computar  $\text{top}$  (matriz de adyacencia), donde  $\text{top}[p,q]$  es true si  $p$  y  $q$  son vecinos

Cada nodo debe ejecutar un nº de rondas para conocer la topología completa. Si el diámetro  $D$  de la red es conocido se resuelve con el siguiente algoritmo

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

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

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

Rara vez se conoce el valor de  $D$

Excesivo intercambio de mensajes los procesos cercanos al “centro” conocen la topología más pronto y no aprenden nada nuevo en los intercambios

El tema de la terminación ¿local o distribuida?

¿Cómo se pueden solucionar estos problemas?

- Despues de  $r$  rondas,  $p$  conoce la topología a distancia  $r$  de él. Para cada nodo  $q$  dentro de la distancia  $r$  de  $p$ , los vecinos de  $q$  estarán almacenados en la fila  $q$  de  $top \Rightarrow p$  ejecutó las rondas suficientes tan pronto como cada fila de  $top$  tiene algún valor true
- Luego necesita ejecutar una última ronda para intercambiar la topología con sus vecinos
- No siempre la terminación se puede determinar localmente

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

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

## Pipeline

Un pipeline es un arreglo lineal de procesos “filtro” que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida

Estos procesos (“workers”) pueden estar en procesadores que operan en paralelo, en un primer esquema a lazo abierto ( $W_1$  en el INPUT,  $W_n$  en el OUTPUT)

Un segundo esquema es el pipeline circular, donde  $W_n$  se conecta con  $W_1$

Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe

En un tercer esquema posible (cerrado), existe un proceso coordinador que maneja la “realimentación” entre  $W_n$  y  $W_1$

Ejemplo: multiplicación de matrices en bloques

## Probe-echo

Arboles 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 (probe-echo) es el análogo concurrente de DFS

Prueba-eco se basa en el envío de un mensajes (“probe”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“echo”)

Los probes se envían en paralelo a todos los sucesores

Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (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: broadcast ch(m);

Los mensajes broadcast de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden

Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida. Ejemplo: semáforos distribuidos, la base es un ordenamiento total de eventos de comunicación mediante el uso de relojes lógicos

## Token passing

Un paradigma de interacción muy usado se basa en un tipo especial de mensaje (“token”) que puede usarse para otorgar un permiso (control) o recoger información global

de la arquitectura distribuida. Un ejemplo del primer tipo de algoritmos es el caso de tener que controlar exclusión mutua distribuida

Ejemplos de recolección de información de estado son los algoritmos de detección de terminación en computación distribuida

Aunque el problema de la SC se da principalmente en programas de MC, puede encontrarse en programas distribuidos cuando hay algún recurso compartido que puede usar un único proceso a la vez

Generalmente es una componente de un problema más grande, tal como asegurar consistencia en un sistema de BD

Soluciones posibles: Monitor activo que da permiso de acceso (ej: locks en archivos), semáforos distribuidos (usando broadcast, con gran intercambio de mensajes), o token ring (descentralizado y fair)

## 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 del 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 descentralizado: 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

Para resolver C/S la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas)

Además, cada cliente necesita un canal de reply distinto...

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

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

## Diferencias entre RPC y rendezvous

Difieren en la manera de servir la invocación de operaciones:

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

## RPC

Los programas se descomponen en módulos (con procesos y procedures (exportables)), que pueden residir en espacios de direcciones distintos

Los procesos de un módulo pueden compartir variables y llamar a procedures de ese módulo

Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste

Los módulos tienen especificación e implementación de procedures:

**module *Mname***

headers de procedures exportados (visibles)

**body**

declaraciones de variables

código de inicialización

cuerpos de procedures exportados

procedures y procesos locales

**end**

Los procesos locales son llamados background para distinguirlos de las operaciones exportadas

Header de un procedure visible:

**op *opname* (formales) [returns *result*]**

El cuerpo de un procedure visible es contenido 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:

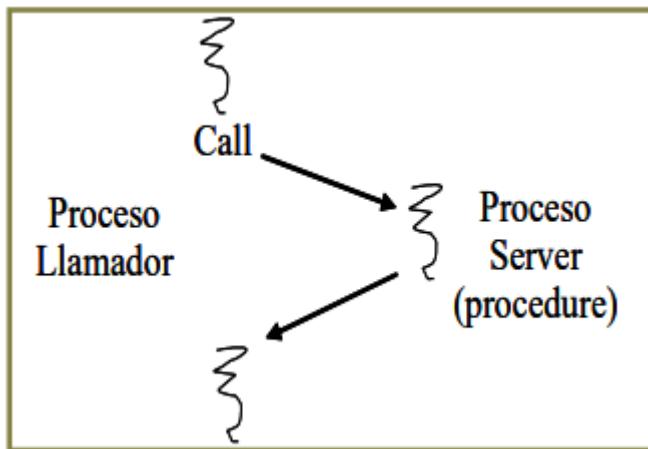
**call *Mname.opname* (argumentos)**

Para un llamado local, el nombre del módulo se puede omitir

La implementación de un llamado intermodulo es distinta que para uno local, ya que los dos módulos pueden estar en distintos espacios: un nuevo proceso sirve el llamado, y los argumentos son pasados como mensajes entre el llamador y el proceso server

El llamador se demora mientras el proceso servidor ejecuta el cuerpo del procedure que implementa opname

Cuando el server vuelve de opname envía los resultados al llamador y termina. Después de recibir los resultados, el llamador sigue



Si el proceso llamador y el procedure están en el mismo espacio de direcciones, es posible evitar crear un nuevo proceso

En general, un llamado será remoto => se debe crear un proceso server o alocarlo de un pool preexistente

## Sincronización de módulos

Por sí mismo, RPC es solo un mecanismo de comunicación

Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador (como si éste estuviera ejecutando el llamado => la sincronización entre ambos es implícita)

Necesitamos que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo)

Esto comprende Exclusión Mutua y Sincronización por Condición

Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan:

- Con exclusión mutua (un solo proceso por vez):
  - Las variables compartidas son protegidas automáticamente contra acceso concurrente, pero es necesario programar sincronización por condición
- Concurrentemente:
  - Necesitamos mecanismos para programar exclusión mutua y sincronización por condición (cada módulo es un programa concurrente) => podemos usar cualquier método ya descrito (semáforos, monitores, o incluso rendezvous)

Es más general asumir que los procesos pueden ejecutar concurrentemente (más eficiente en un multiprocesador de memoria compartida). Asumimos que procesos en un módulo ejecutan concurrentemente, usando por ejemplo time slicing

## Ejemplo cliente/servidor

### Time server

Módulo que brinda servicios de timing a procesos cliente en otros módulos

Dos operaciones visibles: `get_time` y `delay(interval)`

Un proceso interno que continuamente inicia un timer por hardware, luego incrementa el tiempo al ocurrir la interrupción de timer

```

module TimeServer
  op get_time() returns INT;
  op delay(INT interval, INT myid);
body
  INT tod = 0;
  SEM m= 1;
  SEM d[n] = ([n] 0);
  QUEUE of (INT waketime, INT id's) napQ;
  proc get_time () returns time
  { time := tod; }

  proc delay(interval, myid)
  { INT waketime = tod + interval;
    P(m);
    insert ((waketime, myid) napQ);
    V(m);
    P(d[myid]);
  }
end TimeServer;

```

## Manejo de caches en un sistema de archivos distribuido

Versión simplificada de un problema que se da en sistemas de archivos y BD distribuidos

Suponemos procesos de aplicación que ejecutan en una WS, y archivos de datos almacenados en un FS. Los programas de aplicación que quieren acceder a datos del FS, llaman procedimientos `read` y `write` del módulo local `FileCache`. Leen o escriben arreglos de caracteres

Los archivos se almacenan en el FS en bloques de 1024 bytes, fijos. El módulo `FileServer` maneja el acceso a bloques del disco; provee dos procedimientos (`ReadBlk` y `WriteBlk`)

El módulo FileCache mantiene en cache los bloques recientemente leídos. Al recibir pedido de read, FileCache primero chequea si los bytes solicitados están en su cache. Sino, llama al procedimiento readblock del FileServer. Algo similar ocurre con los write

```

Module FileCache  # ubicado en cada workstation
  op read (INT count ; result CHAR buffer[ * ] );
  op write (INT count; CHAR buffer[*] );
body
  cache de N bloques; descripción de los registros de cada file; semáforos para sincronizar acceso al cache;
proc read (count, buffer)
  { IF (los datos pedidos no están en el cache)
    { seleccionar los bloques del cache a usar;
      IF (se necesita vaciar parte del cache) FileServer.writeblk(...);
      FileServer.readblk(...);
    }
    buffer= número de bytes requeridos del cache;
  }
proc write(count, buffer)
  { IF (los datos apropiados no están en el cache)
    { seleccionar los bloques del cache a usar;
      IF (se necesita vaciar parte del cache) FileServer.writeblk(...);
    }
    bloquedeCache= número de bytes desde buffer;
  }
end FileCache;

```

Los llamados de los programas de aplicación de las WS son locales a su FileCache, pero desde estos módulos se invocan los procesos remotos de FileServer

FileCache es un server para procesos de aplicación; FileServer es un server para múltiples clientes FileCache, uno por WS

Si existe un FileCache por programa de aplicación, no se requiere sincronización interna entre los read y write, porque sólo uno puede estar activo. Si múltiples programas de aplicación usaran el mismo FileCache, tendríamos que usar semáforos para implementar la EM en el acceso a FileCache

En cambio en FileServer se requiere sincronización interna, ya que atiende múltiples FileCache y contiene un proceso DiskDriver (la sincronización no se muestra en el código)

```

Module FileServer # ubicado en el servidor
  op readblk (INT fileid, offset; result CHAR blk[1024]);
  op writeblk (INT fileid, offset; CHAR blk[1024]);
body
  cache de bloques; cola de pedidos pendientes; semáforos para acceso al cache y a la cola;
proc readblk (fileid, offset, blk)
  { IF (los datos pedidos no están en el cache) {encola el pedido; esperar que la lectura sea procesada;}
    blk= bloques pedidos del disco;
  }
proc writeblk (fileid, offset, blk)
  { Ubicar el bloque en cache;
    IF (es necesario grabar físicamente en disco) {encola el pedido; esperar que la escritura sea procesada;}
    bloque cache = blk;
  }
process DiskDriver
  { WHILE (true)
    { esperar por un pedido de acceso físico al disco; arrancar una operación física; esperar interrupción;
      despertar el proceso que está esperando completar el request;
    }
  }
end FileServer;

```

## Pares interactuantes

Si dos procesos de diferentes módulos deben intercambiar valores, cada módulo debe exportar un procedimiento que el otro módulo llamará

```

module Intercambio [i = 1 to 2]
  op depositar(int);
body
  int otrovalor;
  sem listo = 0;
proc depositar(otro)
  { otrovalor = otro;
    V(listo);
  }
process Worker
  { int mivalor;
    call Intercambio[3-i].depositar(mivalor);
    P(listo); .....
  }
end Intercambio

```

## RPC en Java (Remote Method Invocation: RMI)

Java soporta el uso de RPC en programas distribuidos mediante la invocación de métodos remotos (RMI)

Una aplicación que usa RMI tiene 3 componentes:

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

El server y los clientes pueden residir en máquinas diferentes

## Rendezvous

RPC por sí mismo sólo brinda un mecanismo de comunicación inter módulo. Dentro de un módulo es necesario programar la sincronización. Además, a veces son necesarios procesos extra sólo para manipular los datos comunicados por medio de RPC (ej: Merge)

Rendezvous combina comunicación y sincronización:

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

La especificación de un módulo contiene declaraciones de los headers de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones

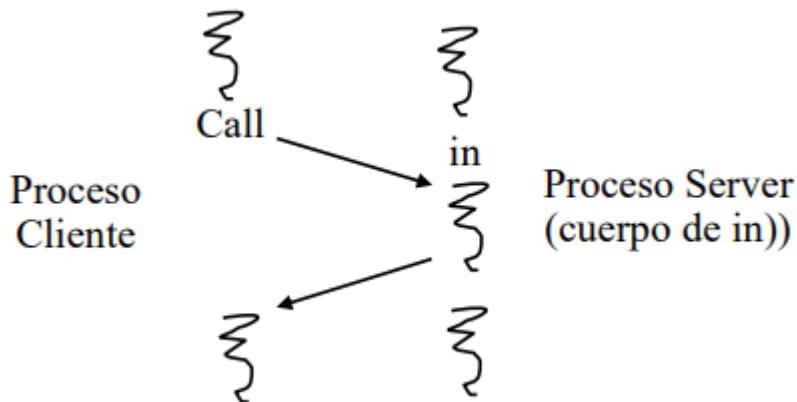
Si un módulo exporta opname, el proceso server en el módulo realiza rendezvous con un llamador de opname ejecutando una sentencia de entrada:

**in opname (parámetros formales) → S; ni**

Las partes entre in y ni se llaman operación guardada

Una sentencia de entrada demora al proceso server hasta que haya al menos un llamado pendiente de opname; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta S y finalmente retorna los parámetros de resultado al llamador. Luego, ambos procesos pueden continuar

A diferencia de RPC el server es un proceso activo



Combinando comunicación guardada con rendezvous:

**in op<sub>1</sub> (formales<sub>1</sub>) and B<sub>1</sub> by e<sub>1</sub> → S<sub>1</sub>;**  
 ...  
 op<sub>n</sub> (formales<sub>n</sub>) and B<sub>n</sub> by e<sub>n</sub> → S<sub>n</sub>;  
**ni**

Los Bi son expresiones de sincronización opcionales (pueden referenciar a los parámetros formales)

Los ei son expresiones de scheduling opcionales. Sirven para modificar el orden en el que se atiendan múltiples pedidos sobre una operación (pueden referenciar a los parámetros formales)

El funcionamiento de la comunicación guardada es:

1. Evaluar todas las condiciones booleanas ( $B_i$ ) (en caso de no tener, se considera true)
2. De las que son verdaderas, ver si hay un llamado pendiente
3. De todas las operaciones resultantes, se elige una en forma no determinística
4. Una vez decidida la operación a ejecutar, se va a elegir para atender el pedido que minimice la expresión de scheduling ( $e_i$ )

Si, al evaluar, todas las condiciones son falsas, se sale de la sentencia in

Si, al evaluar, hay algunas que son verdaderas, pero ninguna tiene llamados pendientes, se espera hasta que haya algún llamado

## Ejemplos

Buffer limitado

```
module BufferLimitado
  op depositar (typeT), retirar (OUT typeT);

  body
    process Buffer
      { queue buf;
        int cantidad = 0;

        while (true)
          { in depositar (item) and cantidad < n -> push (buf, item);
            cantidad = cantidad + 1;
            □ retirar (OUT item) and cantidad > 0 -> pop (buf, item);
            cantidad = cantidad - 1;
            ni
          }
    }
  end BufferLimitado
```

## Filósofos centralizado

```
module Mesa
  op tomar(int), dejar(int);
body
  process Mozo
    { bool comiendo[5] =([5] false);
      while (true)
        in tomar(i) and not (comiendo[izq(i)] or comiendo[der(i)]) → comiendo[i] = true;
        □ dejar(i) → comiendo[i] = false;
        ni
    }
end Mesa

module Persona [i = 0 to 4]
Body
  process Filosofo
    { while (true)
      { call Mesa.tomar(i);
        come;
        call Mesa.dejar(i);
        piensa;
      }
    }
}
```

Nos puede traer problemas ya que, por ejemplo, si el filósofo 3 no puede comer porque está comiendo el filósofo 4, pero detrás de él viene el filósofo 2, que sí puede comer, no podrá hacerlo, porque el filósofo 3 está por delante

## Time server

A diferencia del ejemplo visto para RPC, waketime hace referencia a la hora que debe despertarse

```

module TimeServer
  op get_time (OUT int);
  op delay (int);
  op tick ();

body TimeServer
  process Timer
    { int tod = 0;
      while (true)
        in get_time (OUT time) → time = tod;
         delay (waketime) and waketime <= tod by waketime → skip;
         tick () → tod = tod + 1; reiniciar timer;
        ni
    }
end TimeServer

```

Alocador SJN

```

module Alocador_SJN
  op pedir(int), liberar();

body
  process SJN
    { bool libre = true;
      while (true)
        in pedir (tiempo) and libre by tiempo → libre = false;
         liberar () → libre = true;
        ni
    }
end SJN_Allocator

```

## Rendezvous en ADA

Desarrollado por el Departamento de Defensa de USA para que sea el estándar en programación de aplicaciones de defensa (desde sistemas de Tiempo Real a grandes sistemas de información)

Desde el punto de vista de la concurrencia, un programa Ada tiene tasks (tareas) que pueden ejecutar independientemente y contienen primitivas de sincronización

Los puntos de invocación (entrada) a una tarea se denominan entrys y están especificados en la parte visible (header de la tarea)

Una tarea puede decidir si acepta la comunicación con otro proceso, mediante la primitiva accept

Se puede declarar un type task, y luego crear instancias de procesos (tareas) identificado con dicho tipo (arreglo, puntero, instancia simple)

## Tasks

➤ La forma más común de especificación de task es:

**TASK** *nombre* **IS**  
declaraciones de ENTRYs  
**end;**

➤ La forma más común de cuerpo de task es:

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

Una especificación de TASK define una única tarea

Una instancia del correspondiente task body se crea en el bloque en el cual se declara el TASK

## Sincronización: Call

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

Entry:

- Declaración de entry simples y familia de entry (parámetros IN, OUT y IN OUT)
- Entry call: la ejecución demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción)

### **Tarea.entry (parámetros)**

- Entry call condicional:

- Si un pedido es aceptado inmediatamente, realiza el entry call y realiza las sentencias adicionales

- Si un pedido no es aceptado inmediatamente, realiza las sentencias

```
select entry call;
      sentencias adicionales;
else
      sentencias;
end select;
```

- Entry call temporal:

- Lo mismo que el anterior pero espera un tiempo antes de ir a realizar las sentencias

```
select entry call;
      sentencias adicionales;
or delay tiempo
      sentencias;
end select;
```

### Sincronización: 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  $\Rightarrow$  accept E1; sentencias1
or      ...
or      when Bn  $\Rightarrow$  accept En; sentenciasn
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

## Ejemplos

Mailbox para 1 mensaje

```

TASK TYPE Mailbox IS
  ENTRY Depositar (msg: IN mensaje);
  ENTRY Retirar (msg: OUT mensaje);
END Mailbox;

A, B, C : Mailbox;

TASK BODY Mailbox IS
  dato: mensaje;
BEGIN
  LOOP
    ACCEPT Depositar (msg: IN mensaje) DO dato := msg; END Depositar;
    ACCEPT Retirar (msg: OUT mensaje) DO msg := dato; END Retirar;
  END LOOP;
END Mailbox;

```

Podemos utilizar estos mailbox para manejar mensajes: A.Depositar(x1);  
 B.Depositar(x2); C.Retirar(x3);

## Lectores-escritores

### **Procedure Lectores-Escritores is**

Task *Sched* IS

Entry *InicioLeer*;  
Entry *FinLeer*;  
Entry *InicioEscribir*;  
Entry *FinEscribir*;

End *Sched*;

Task type *Lector*;

Task body *Lector* is

Begin

Loop

*Sched.InicioLeer*; ... *Sched.FinLeer*;

End loop;

End *Lector*;

Task type *Escritor*;

Task body *Escritor* is

Begin

Loop

*Sched.InicioEscribir*; ... *Sched.FinEscribir*;

End loop;

End *Lector*;

*VecLectores*: array (1..cantL) of *Lector*;

*VecEscritores*: array (1..cantE) of *Escritor*;

Task body *Sched* is

*numLect*: integer := 0;

    Begin

        Loop

            Select

                When *InicioEscribir*'Count = 0 =>

                    accept *InicioLeer*;

*numLect* := *numLect*+1;

                or accept *FinLeer*;

*numLect* := *numLect*-1;

                or When *numLect* = 0 =>

                    accept *InicioEscribir*;

                    accept *FinEscribir*;

                    For *i* in 1..*InicioLeer*'count loop

                        accept *InicioLeer*;

*numLect* := *numLect*+1;

                    End loop;

                End select;

            End loop;

    End *Sched*;

    Begin

        Null;

**End Lectores-Escritores**

Mailbox para N mensajes (buffer limitado)

Con una cola:

```
TASK Mailbox IS
  ENTRY Depositar (msg: IN mensaje);
  ENTRY Retirar (msg: OUT mensaje);
END Mailbox;

TASK BODY Mailbox IS
  buf: queue;
  cantidad integer := 0;
BEGIN
  LOOP
    SELECT
      WHEN cantidad < N => ACCEPT Depositar (msg: IN mensaje) DO
        push (buf, msg);
        cantidad := cantidad +1;
      END Depositar;
    OR
      WHEN cantidad > 0 => ACCEPT Retirar (msg: OUT mensaje) DO
        pop (buf, msg);
        cantidad = cantidad -1;
      END Retirar;
    END SELECT;
  END LOOP;
END Mailbox;
```

Con un arreglo:

```
TASK Mailbox IS
  ENTRY Depositar (msg: IN mensaje);
  ENTRY Retirar (msg: OUT mensaje);
END Mailbox;

TASK BODY Mailbox IS
  datos: array (0..N-1) of mensaje;
  cant, pri, ult integer := 0;
BEGIN
  LOOP
    SELECT
      WHEN cant < N => ACCEPT Depositar (msg: IN mensaje) DO
        ult := (ult+1) MOD N; datos[ult] := msg; cant := cant +1;
      END Depositar;
    OR
      WHEN cant > 0 => ACCEPT Retirar (msg: OUT mensaje) DO
        msg := datos[pri]; pri := (pri+1) MOD N; cant := cant -1;
      END Retirar;
    END SELECT;
  END LOOP;
END Mailbox;
```

Filósofos centralizado

Múltiples entry

```
TASK Mesa IS
  ENTRY Tomar0; ENTRY Tomar1; ENTRY Tomar2; ENTRY Tomar3; ENTRY Tomar4;
  ENTRY Dejar (id: IN integer);
END Mesa;

TASK BODY Mesa IS
  Comiendo: array (0..4) of bool := (0..4=> false);
  BEGIN
    For i in 0..4 loop
      Filosofos(i).identificacion(i);
    end loop;
    LOOP
      SELECT
        when (not (comiendo(4) or comiendo(1))) => ACCEPT Tomar0; comiendo(0) := true;
        OR when (not (comiendo(0) or comiendo(2))) => ACCEPT Tomar1; comiendo(1) := true;
        OR when (not (comiendo(1) or comiendo(3))) => ACCEPT Tomar2; comiendo(2) := true;
        OR when (not (comiendo(2) or comiendo(4))) => ACCEPT Tomar3; comiendo(3) := true;
        OR when (not (comiendo(3) or comiendo(0))) => ACCEPT Tomar4; comiendo(4) := true;
        OR ACCEPT Dejar(id: IN integer) do
          comiendo(id) := false;
        end Dejar;
      END SELECT;
    END LOOP;
  END Mesa;
```

```
TASK TYPE Filosofo IS
  ENTRY Identificacion (ident: IN integer);
END Filosofo;

Filosofos: array (0..4) of Filosofo;

TASK BODY Filosofo IS
  id: integer;
  BEGIN
    ACCEPT Identificacion (ident : IN integer) do
      id := ident;
    End Identificacion;
    LOOP
      if ( id = 0) then Mesa.Tomar0;
      else if ( id = 1) then Mesa.Tomar1;
      else if ( id = 2) then Mesa.Tomar2;
      else if ( id = 3) then Mesa.Tomar3;
      else if ( id = 4) then Mesa.Tomar4;
      //Come
      Mesa.Dejar(id);
      //Piensa
    END LOOP;
  END Mesa;
```

Encolar pedidos:

```
TASK TYPE Filosofo IS
    ENTRY Identificacion (ident: IN integer);
    ENTRY Comer;
END Filosofo;
TASK Mesa IS
    ENTRY Tomar (id: IN integer);
    ENTRY Dejar (id: IN integer);
END Mesa;

Filosofos: array (0..4) of Filosofo;

TASK BODY Filosofo IS
    id: integer;
BEGIN
    ACCEPT Identificacion (ident : IN integer) do id := ident; End Identificacion;
    LOOP
        Mesa.Tomar(id);
        Accept Comer;
        //Come
        Mesa.Dejar(id);
        //Piensa
    END LOOP;
END Mesa;
```

```
TASK BODY Mesa IS
    Comiendo: array (0..4) of bool := (0..4=> false);
    QuiereC: array (0..4) of bool := (0..4=> false);
    aux: integer;
BEGIN
    For i in 0..4 loop  Filosofos(i).identificacion(i);  end loop;
    LOOP
    SELECT
        ACCEPT Tomar(id: IN integer) do  aux := id; END Tomar;
        if (not (comiendo((aux+1) mod 5) or comiendo((aux-1) mod 5)) ) then
            comiendo(aux) = true;
            Filosofos(aux).Comer;
        else QuiereC (aux) := true;  end if;
    OR ACCEPT Dejar(id: IN integer) do  aux := id; end Dejar;
        comiendo(aux) := false;
        for i in 0..4 loop
            if (QuiereC(i) and not (comiendo((i+1) mod 5) or comiendo((i-1) mod 5)) ) then
                comiendo(i) = true;
                QuiereC(i) := false;
                Filosofos(i).Comer;
            end if;
        end loop;
    END SELECT;
    END LOOP;
END Mesa;
```

Time server

```
PROCEDURE DESPERTADORES IS
Task TimeServer is
    entry get_time (hora: OUT int);    entry delay (hd, id: IN int);  entry tick;
End TimeServer;
Task Reloj;
Task Type Cliente Is
    entry Identificar (identificacion: IN integer);    entry seguir;
End Cliente;
ArrClientes: array (1..C) of Cliente;
Task Body Cliente Is
    id: integer;  hora: integer;
BEGIN
    ACCEPT Identificar (identificacion : IN integer) do id := identificación;  End Identificar;
    TimeServer.get_time(hora);
    TimeServer.delay(hora+....., id);
    ACCEPT seguir;
End Cliente;
Task Body Reloj is
BEGIN
    loop    delay(1);  TimeServer.tick;  end loop;
End Reloj;
```

```
Task Body TimeServer is
    actual: integer := 0;
    dormidos: colaOrdenada;
    auxId, auxHora: integer;
BEGIN
    LOOP
        SELECT
            when (tick'count =0) =>ACCEPT get_time (hora: OUT integer) do hora := actual; END get_time;
            OR when (tick'count =0) =>ACCEPT delay(hd, id: IN integer) do agregar(dormidos, (id,hd)); END delay;
            OR ACCEPT tick;
                actual := actual +1;
                while (not empty (dormidos)) and then (VerHoraPrimero(dormidos) <= actual)) loop
                    sacar(dormidos, (auxId, auxHora));
                    ArrClientes(auxId).seguir;
                end loop;
            END SELECT;
        END LOOP;
End TimeServer;

BEGIN
    for i in 1..C loop
        ArrClientes(i).identificacion(i);
    end loop;
End DESPERTADORES;
```

## Alocador SJN

```
PROCEDURE SchedulerSJN IS
  Task Alocador_SJN is
    entry pedir (tiempo, id: IN integer);
    entry liberar;
  End Alocador_SJN ;

  Task Type Cliente Is
    entry Identificar (identificacion: IN integer);
    entry usar;
  End Cliente;
  ArrClientes: array (1..C) of Cliente;
  Task Body Cliente Is
    id: integer; tiempo: integer;
  BEGIN
    ACCEPT Identificar (identificacion : IN integer) do
      id := identificación;
    End Identificar;
    loop
      //trabaja y determina el valor de tiempo
      Alocador_SJN.pedir(id, tiempo);
      Accept usar;
      //Usa el recurso
      Alocador_SJN.liberar;
    end loop;
  End Cliente;
```

```
Task Body Alocador_SJN is
  libre: boolean := true;
  espera: colaOrdenada;
  tiempo, aux: integer;
  Begin
    loop
      aux := -1;
      select
        accept Pedir (tiempo, id: IN integer) do
          if (libre) then libre:= false; aux := id;
          else agregar(espera, (id, tiempo)); end if;
        end Pedir;
        or accept liberar;
          if (empty (espera)) then libre := true;
          else sacar(espera, (aux, tiempo)); end if;
        end select;
        if (aux <> -1) then ArrClientes(aux).usar; end if;
      end loop;
  End Alocador_SJN ;

  BEGIN
    for i in 1..C loop
      ArrClientes(i).identificacion(i);
    end loop;
  END SchedulerSJN;
```

# Clase 9 - Pthreads y MPI

## Pthreads

Thread: proceso “liviano” que tiene su propio contador de programa y su pila de ejecución, pero no controla el “contexto pesado” (por ejemplo, las tablas de página)

Pthreads es una biblioteca para programación paralela en memoria compartida, se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

Una función reentrante es aquella que, a pesar de ser llamadas por más de un proceso al mismo tiempo, no generará inconvenientes al no trabajar con variables comunes

### Creación de un hilo

```
int pthread_create (pthread_t *thread_handle, const pthread_attr_t *attribute,  
                    void * (*thread_function)(void *), void *arg);
```

Para hacer referencia al hilo se hará referencia al manejador

Una vez creado, el hilo ejecuta la función thread\_function con los argumentos arg

### Terminación de un hilo

```
int pthread_exit (void *res);
```

Con esta función se puede obtener un resultado de la ejecución del hilo una vez terminada

### Esperar a que un hilo termine

```
int pthread_join (pthread_t thread, void **ptr);
```

## Cancelar un hilo

```
int pthread_cancel (pthread_t thread);
```

Hasta que todos los hilos no hayan terminado, el main debe esperar para finalizar

## Exclusión mutua

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

```
int pthread_mutex_lock ( pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

```
int pthread_mutex_init (pthread_mutex_t *mutex,  
                      const pthread_mutexattr_t *lock_attr);
```

## Productores/Consumidores

Un thread productor no debe sobrescribir el buffer compartido cuando el elemento anterior no ha sido tomado por un thread consumidor

Un thread consumidor no puede tomar nada de la estructura compartida hasta no estar seguro de que se ha producido algo anteriormente

Los consumidores deben excluirse entre sí

Los productores deben excluirse entre sí

En este ejemplo el buffer es de tamaño 1

```
pthread_mutex_t mutex;
int hayElemento;
tipo_elemento Buffer;
...
main()
{ hayElemento= 0;
  pthread_init ();
  pthread_mutex_init(&mutex, NULL);

  /* Create y join de threads productores y consumidores*/
}
```

```
void *productor (void *datos)
{ tipo_elemento elem;
  int ok;
  ....
  while (true)
  { ok = 0;
    generar_elemento(&elem);
    while (ok == 0)
    { pthread_mutex_lock(&mutex);
      if (hayElemento== 0)
      { Buffer = elem;
        hayElemento = 1;
        ok = 1;
      }
      pthread_mutex_unlock(&mutex);
    }
  }
}
```

```

void *consumidor(void *datos)
{
    int ok;
    tipo_elemento elem;
    ....
    while (true)
    {
        ok = 0;
        while (ok == 0)
        {
            pthread_mutex_lock(&mutex);
            if (hayElemento == 1)
            {
                elem = Buffer;
                hayElemento = 0;
                ok = 1;
            }
            pthread_mutex_unlock(&mutex);
        }
        procesar_elemento(elem);
    }
}

```

## Tipos de mutex

Pthreads soporta tres tipos de mutexs (Locks): Normal, Recursive y Error Check

- Un Mutex con el atributo Normal NO permite que un thread que lo tiene bloqueado vuelva a hacer un lock sobre él (deadlock)
- Un Mutex con el atributo Recursive SI permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él. Simplemente incrementa una cuenta de control
- Un Mutex con el atributo ErrorCheck responde con un reporte de error al intento de un segundo bloqueo por el mismo thread

El tipo de Mutex puede setearse entre los atributos antes de su inicialización

## Overhead de bloqueos por exclusión mutua

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

A menudo se puede reducir el overhead por espera ociosa, utilizando la función `pthread_mutex_trylock` la cual retorna el control informando si pudo hacer o no el lock

**`int pthread_mutex_trylock (pthread_mutex_t *mutex_lock).`**

Evita tiempos ociosos

Menos costoso por no tener que manejar las colas de espera

## Variables de condición

Podemos utilizar variables de condición para que un thread se autobloquee hasta que se alcance un estado determinado del programa

Cada variable de condición estará asociada con un predicado. Cuando el predicado se convierte en verdadero (TRUE) la variable de condición da una señal para el/los threads que están esperando por el cambio de estado de la condición

Una única variable de condición puede asociarse a varios predicados (difícil el debug)

Una variable de condición siempre tiene un mutex asociado 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)

```

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)

```

pthread\_cond\_wait duerme a un proceso dentro de una sección crítica. La función va a desbloquear el mutex, dormir al proceso, cuando sea despertado, obtener el lock y salir de la función

### Productores/Consumidores

```

pthread_cond_t vacio, lleno;
pthread_mutex_t mutex;
int hayElemento;
tipo_elemento Buffer;
...
main()
{
    ...
    hayElemento= 0;
    pthread_init();
    pthread_cond_init(&vacio, NULL);
    pthread_cond_init(&lleno, NULL);
    pthread_mutex_init(&mutex, NULL);
    ...
}

```

```

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

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

```

```

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

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

```

## Atributos

La API Pthreads permite que se pueda cambiar los atributos por defecto de las entidades, utilizando **attributes objects**

Un attribute object es una estructura de datos que describe las propiedades de la entidad en cuestión (thread, mutex, variable de condición)

Una vez que estas propiedades están establecidas, el attribute object es pasado al método que inicializa la entidad

Ventajas

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

## Semáforos

### Declaración y operaciones

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 ( $\neq 0$ )

sem\_wait(&semaforo) → equivale al P

sem\_post(&semaforo) → equivale al V

Existen funciones extras para: wait condicional, obtener el valor de un semáforo y destruir un semáforo (ESTE TIPO DE FUNCIONES EXTRAS NO SE PUEDEN USAR EN LA PRÁCTICA DE LA MATERIA)

### Productor consumidor

Las funciones de Productor y Consumidor serán ejecutadas por threads independientes

Acceden a un buffer compartido (datos)

El productor deposita una secuencia de enteros de 1 a numItems en el buffer

El consumidor busca estos valores y los suma

Los semáforos vacío y lleno garantizan el acceso alternativo de productor y consumidor sobre el buffer

```
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1

void *Productor(void *);  
void *Consumidor(void *);  
  
sem_t vacio, lleno;  
int dato, numItems;
```

```
int main(int argc, char * argv[ ])  
{  
    .....  
    sem_init (&vacio, SHARED, 1);  
    sem_init (&lleno, SHARED, 0);  
    .....  
    pthread_create (&pid, &attr, Productor, NULL);  
    pthread_create (&cid, &attr, Consumidor, NULL);  
    pthread_join (pid, NULL);  
    pthread_join (cid, NULL);  
}
```

```

void *Productor (void *arg)
{ int item;
  for (item = 1; item <= numItems; item++)
  { sem_wait(&vacío);
    dato = item;
    sem_post(&lleno);
  }
  pthreads_exit();
}

void *Consumidor (void *arg)
{ int total = 0, item, aux;
  for (item = 1; item <= numItems; item++)
  { sem_wait(&lleno);
    aux = dato;
    sem_post(&vacío);
    total = total + aux;
  }
  printf("TOTAL: %d\n", total);
  pthreads_exit();
}

```

## Monitores

Pthreads nos permite manejar la Exclusión Mutua por medio de las variables mutex

Pthreads nos permite manejar la Sincronización por Condición utilizando variables condición para que un thread se auto bloquee hasta que se alcance un estado determinado del programa. Una variable de condición siempre tiene un mutex asociada a ella

Pthreads no posee “Monitores”, pero con las dos herramientas que mencionamos se puede 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

### Lectores y escritores

```
#include <pthread.h>

void *Escritor(void *);
void *Lector(void *);

int main(int argc, char * argv[ ])
{ int nr = 0, nw = 0, dr = 0, dw = 0, i;
  pthread_cond_t ok_leer, ok_escribir;
  pthread_t lectores[L], escritores[E];
  .....
  pthread_init();
  pthread_cond_init(&ok_leer, NULL);
  pthread_cond_init(&ok_escribir, NULL);
  .....
  for (i=0; i<E;i++) pthread_create (&escritores[i], &attr, Escritor, NULL);
  for (i=0; i<L;i++) pthread_create (&lectores[i], &attr, Lector, NULL);
}
```

```
#include <pthread.h>

void *Escritor(void *);
void *Lector(void *);

int main(int argc, char * argv[ ])
{ int nr = 0, nw = 0, dr = 0, dw = 0, i;
  pthread_cond_t ok_leer, ok_escribir;
  pthread_t lectores[L], escritores[E];
  pthreads_mutex_t mutex;
  .....
  pthread_init();
  pthread_cond_init(&ok_leer, NULL);
  pthread_cond_init(&ok_escribir, NULL);
  pthread_mutex_init(&mutex, NULL);
  .....
  for (i=0; i<E;i++) pthread_create (&escritores[i], &attr, Escritor, NULL);
  for (i=0; i<L;i++) pthread_create (&lectores[i], &attr, Lector, NULL);
}
```

```

void *lector (void*)
{ while (true)
    { pthread_mutex_lock (&mutex);
      Controlador.pedido_leer();
      pthread_mutex_unlock (&mutex);
      //Leer sobre la BD
      pthread_mutex_lock (&mutex);
      Controlador.libera_leer();
      pthread_mutex_unlock (&mutex);
    }
}

```

```

void *escritor (void*)
{ while (true)
    { pthread_mutex_lock (&mutex);
      Controlador.pedido_escribir();
      pthread_mutex_unlock (&mutex);
      //Leer sobre la BD
      pthread_mutex_lock (&mutex);
      Controlador.libera_escribir ();
      pthread_mutex_unlock (&mutex);
    }
}

```

```

void *escritor (void*)
{ while (true)
    { pthread_mutex_lock (&mutex);
      if (nr>0 OR nw>0)
        { dw = dw +1;
          pthread_cond_wait (& ok_escribir, &mutex);
        }
      else nw = nw + 1;
      pthread_mutex_unlock (&mutex);
      //Escribe sobre la BD
      pthread_mutex_lock (&mutex);
      if (dw > 0)
        { dw = dw -1;
          pthread_cond_signal(&ok_escribir);
        }
      else
        { nw = nw -1;
          if (dr > 0)
            { nr = dr;
              dr = 0;
              pthread_cond_broadcast(&ok_leer);
            };
        };
      pthread_mutex_unlock (&mutex);
    };
    pthreads_exit();
};

```

```

void *lector(void*)
{ while (true)
    { pthread_mutex_lock (&mutex);
      if (nw>0)
        { dr = dr +1;
          pthread_cond_wait (& ok_leer, &mutex);
        }
      else nr = nr + 1;
      pthread_mutex_unlock (&mutex);
      //Leer sobre la BD
      pthread_mutex_lock (&mutex);
      nr = nr - 1;
      if (nr == 0 and dw > 0)
        { dw = dw -1;
          pthread_cond_signal(&ok_escribir);
          nw = nw + 1;
        };
      pthread_mutex_unlock (&mutex);
    };
    pthreads_exit();
};

```

## MPI

Send y receive

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

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

## Comunicación bloqueante

Para asegurar la semántica del SEND → no devolver el control del Send hasta que el dato a transmitir esté seguro (Send bloqueante)

Ociosidad del proceso

Hay dos posibilidades:

- Send/Receive bloqueantes sin buffering (similar a PMS). El dato a transmitir está seguro al llegar al ser recibido por el receptor
- Send/Receive bloqueantes con buffering (similar a PMA):
  - El dato a transmitir está seguro al quedar en el buffer
  - Deberá haber un espacio de memoria para el buffer (en general en la máquina receptora)
  - Puede ser más lenta que el bloqueante sin buffering en el caso en que el receptor sea rápido y ya esté en el receive cuando el mensaje llega

## Comunicación no bloqueante

Para evitar overhead (ociosidad o manejo de buffer) se devuelve el control de la operación inmediatamente

Requiere un posterior chequeo para asegurarse la finalización de la comunicación

Deja en manos del programador asegurar la semántica del SEND

Hay dos posibilidades:

- Send/Receive no bloqueantes sin buffering:
  - Hasta que el otro proceso no hace el receive no se hace la comunicación. Hasta ese momento, el dato podría ser modificado
- Send/Receive no bloqueantes con buffering:
  - El dato está seguro al quedar en el buffer
  - El tiempo en el que el dato no está seguro es menor

## Inicio y finalización en MPI

`MPI_Init`: se invoca en todos los procesos antes que cualquier otro llamado a rutinas MPI

Sirve para inicializar el entorno MPI

## **MPI\_Init (int \*argc, char \*\*argv)**

Algunas implementaciones de MPI requieren argc y argv para inicializar el entorno

**MPI\_Finalize:** se invoca en todos los procesos como último llamado a rutinas MPI  
Sirve para cerrar el entorno MPI

## **MPI\_Finalize ()**

## Comunicadores

Son grupos de procesos. Dentro de un comunicador los procesos van a poder comunicarse entre ellos

Un comunicador define el dominio de comunicación

Cada proceso puede pertenecer a muchos comunicadores

Existe un comunicador que incluye a todos los procesos de la aplicación  
**MPI\_COMM\_WORLD**

Son variables del tipo **MPI\_Comm** → almacena información sobre qué procesos pertenecen a él

En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar

## Adquisición de información

**MPI\_Comm\_size:** indica la cantidad de procesos en el comunicador

## **MPI\_Comm\_size (MPI\_Comm comunicador, int \*cantidad)**

**MPI\_Comm\_rank:** indica el “rank” (identificador) del proceso dentro de ese comunicador

## **MPI\_Comm\_rank (MPI\_Comm comunicador, int \*rank)**

rank es un valor entre [0..cantidad]

Cada proceso puede tener un rank diferente en cada comunicador

```
EJEMPLO: #include <mpi.h>

main(int argc, char *argv[])
{   int cantidad, identificador;

    MPI_Init(&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &cantidad);
    MPI_Comm_rank(MPI_COMM_WORLD, &identificador);
    printf("Soy %d de %d \n", identificador, cantidad);
    MPI_Finalize();
}
```

## Tipos de datos para las comunicaciones

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

## Comunicación punto a punto

Diferentes protocolos para Send

- Send bloqueantes con buffering (Bsend)

- Send bloqueantes sin buffering (Ssend)
- Send no bloqueantes (Isend)
- send:
  - Es el send estándar de la librería usada
  - Podría ser cualquiera de los otros 3 o una combinación de ellos

Diferentes protocolos para Recv

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

Bloqueante

`MPI_Send, MPI_Ssend, MPI_Bsend`: rutina básica para enviar datos a otro proceso:

`MPI_Send (void *buf, int cantidad, MPI_Datatype tipoDatos, int destino, int tag, MPI_Comm comunicador)`

Valor de tag entre 0 y `MPI_TAG_UB`

`MPI_Recv`: rutina básica para recibir datos a otro proceso:

`MPI_Recv (void *buf, int cantidad, MPI_Datatype tipoDatos, 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 tipoDatos, int *cantidad)`

```

#include <mpi.h>
main (INT argc, CHAR *argv [ ] ) {
    INT id;
    INT valor, otroValor;
    MPI_status estado;

    MPI_Init (&argc, &argv);
    MPI_Comm_Rank (MPI_COMM_WORLD, &id);
    IF (id == 0) { valor = 14;
        MPI_send (&valor, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
        MPI_recv (&otroValor, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &estado);
    }
    ELSE { valor = 25;
        MPI_recv (&otroValor, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &estado);
        MPI_send (&valor, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    printf ("process %d received a %d\n", id, otroValor);
    MPI_Finalize ( );
}

```

No bloqueante

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 tipoDatos, int destino, int tag, MPI\_Comm comunicador, MPI\_Request \*solicitud)

**MPI\_Irecv** (void \*buf, int cantidad, MPI\_Datatype tipoDatos, 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

```

EJEMPLO: main (int argc, char *argv[])
{
    int cant, id, *dato, i;
    MPI_Status estado;
    MPI_Request req;

    dato = (int *) malloc (100 * sizeof(int));
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == 0)
    {
        cant = atoi(argv[1]);
        //INICIALIZA dato
        MPI_Isend(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD, &req);
        //TRABAJA
        MPI_Wait(&req, &estado);
        for (i=0; i< 100; i++) dato[i]=0;
    }
    else
    {
        MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
        MPI_Get_count(&estado, MPI_INT, &cant);
        //PROCESA LOS DATOS;
    };
    MPI_Finalize;
}

```

```

EJEMPLO: main (int argc, char *argv[])
{
    int id, *dato, i, flag;
    MPI_Status estado;
    MPI_Request req;

    dato = (int *) malloc (100 * sizeof(int));
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == 0)
    {
        //INICIALIZA dato
        MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
    }
    else
    {
        MPI_Irecv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD ,&req);
        MPI_Test(&req, &flag,&estado);
        while (!flag)
        {
            //Trabaja mientras espera
            MPI_Test(&req, &flag,&estado);
        };
        //PROCESA LOS DATOS;
    };
    MPI_Finalize;
}

```

## 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

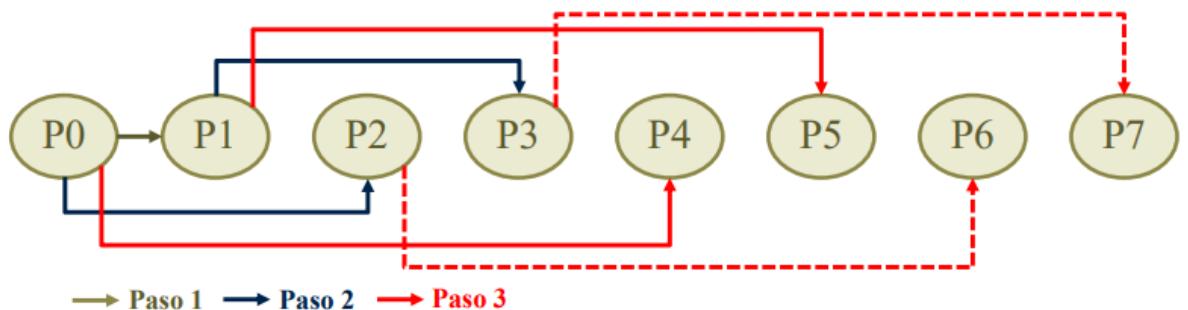
MPI provee un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociado con un comunicador

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`
- Etc.

## Ejemplo de broadcast

### Ventajas del uso de comunicaciones colectivas.



## MPI reduce

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 tipoDatos, MPI_Op operación, int destino, MPI_Comm comunicador)
```

Proceso 0	█	█	█	█
Proceso 1	█	█	█	█
Proceso 2	█	█	█	█
Proceso 3	█	█	█	█

Reduce a 0

Proceso 0	█	█	█	█
Proceso 1				
Proceso 2				
Proceso 3				

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

## MPI gather

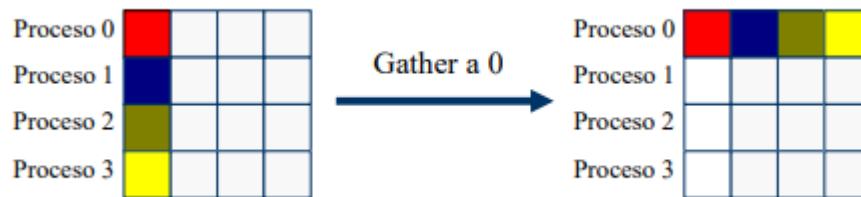
Gather: recolecta el vector de datos de todos los procesos (inclusive el destino) y los concatena en orden para dejar el resultado en un único proceso

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)
```



MPI scatter

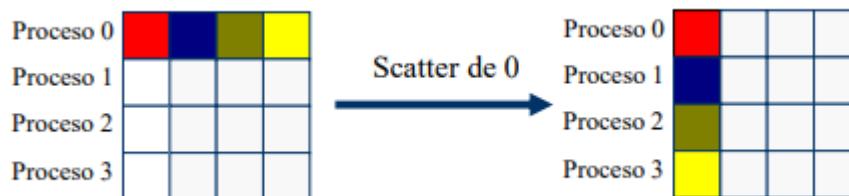
Scatter: reparte un vector de datos entre todos los procesos (inclusive el mismo dueño del vector)

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)
```



## Minimizar los overheads de comunicación

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

# Clase 10 - Introducción a la programación paralela

## Clasificación de arquitecturas paralelas

### Por el espacio de direcciones

Las arquitecturas paralelas se clasifican según su espacio de direcciones en:

- Memoria Compartida
- Memoria Distribuida

Esta clasificación se relaciona con el modelo de comunicación a utilizar:

- Accesos a Memoria Compartida (en memoria compartida)
- Intercambio de mensajes (principalmente memoria distribuida)

En algunos casos también tenemos en la misma plataforma ambos mecanismos

### Multiprocesadores de memoria compartida

Interacción modificando datos en la memoria compartida

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

### Multiprocesadores con memoria distribuida

Procesadores conectados por una red

Memoria local (no hay problemas de consistencia)

La interacción es sólo por pasaje de mensajes

Grado de acoplamiento de los procesadores:

- Multicomputadores (máquinas fuertemente acopladas):
  - Procesadores y red físicamente cerca
  - Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores
  - Alto ancho de banda y velocidad

- Memoria compartida distribuida
- Clusters
- Redes (multiprocesador débilmente acoplado)

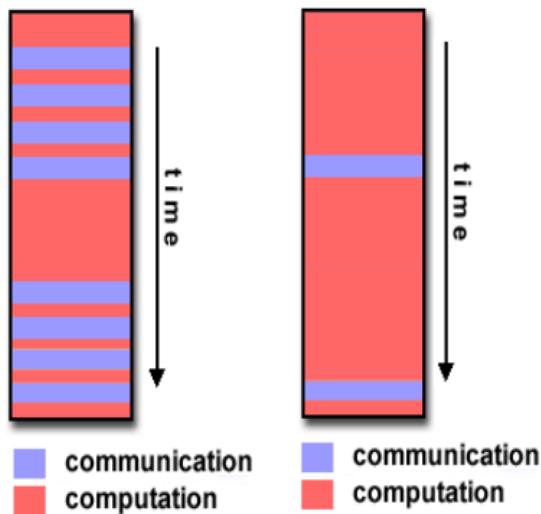
## Por granularidad

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

Relación y adaptación a la arquitectura

Grano fino: cuando hay poco código y luego comunicación, una y otra vez

Grano grueso: cuando hay mucho código y luego comunicación, una y otra vez



Por ejemplo, en un clúster, conviene granularidad gruesa, porque la comunicación es costosa, pero el cómputo no

En un multicore conviene una solución de grano fino, porque tiene una comunicación veloz y un cómputo no tan bueno

## Por el mecanismo de control

Se basa en la manera en que las instrucciones son ejecutadas sobre los datos

Clasifica las arquitecturas en 4 clases:

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

## SISD

### Single Instruction Single Data

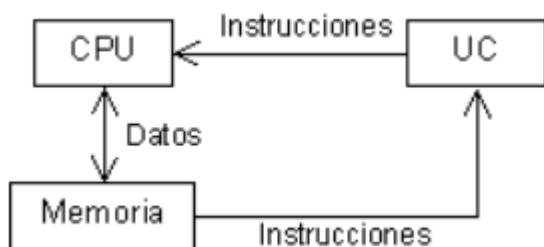
Instrucciones ejecutadas en secuencia, una por ciclo de instrucción

La memoria afectada es usada sólo por ésta instrucción

Usada por la mayoría de los uni procesadores

La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memoria recibe y almacena datos en las escrituras, y brinda datos en las lecturas

Ejecución determinística



## SIMD

### Single Instruction Multiple Data

Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos

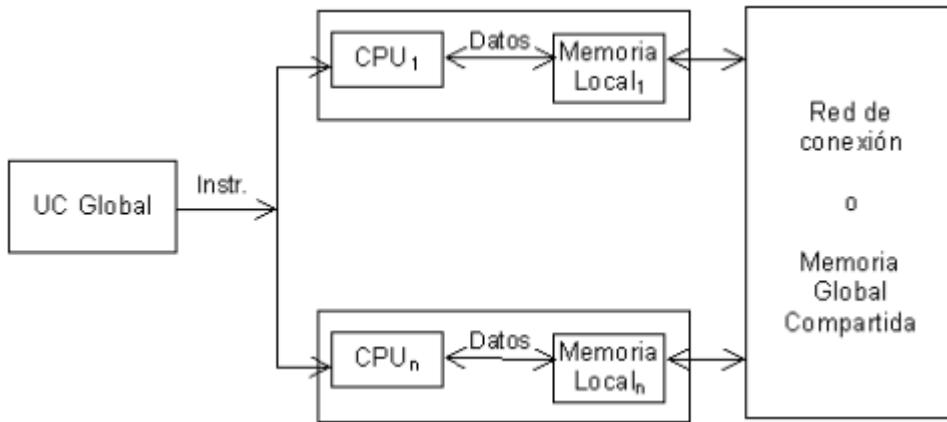
Los procesadores en general son muy simples

El host hace broadcast de la instrucción

Ejecución sincrónica y determinística

Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones

Adecuados para aplicaciones con alto grado de regularidad, (por ejemplo procesamiento de imágenes)



## MISD

Multiple Instruction Single Data

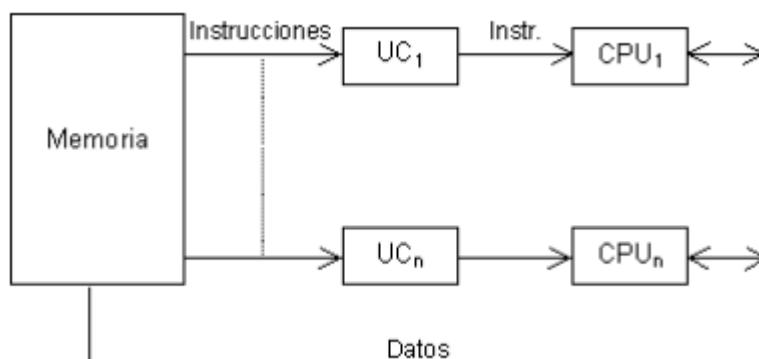
Los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes

Operación sincrónica (en lockstep)

No son máquinas de propósito general (“hipotéticas”, Duncan)

Ejemplos posibles:

- Múltiples filtros de frecuencia operando sobre una única señal
- Múltiples algoritmos de criptografía intentando crackear un único mensaje codificado



## MIMD

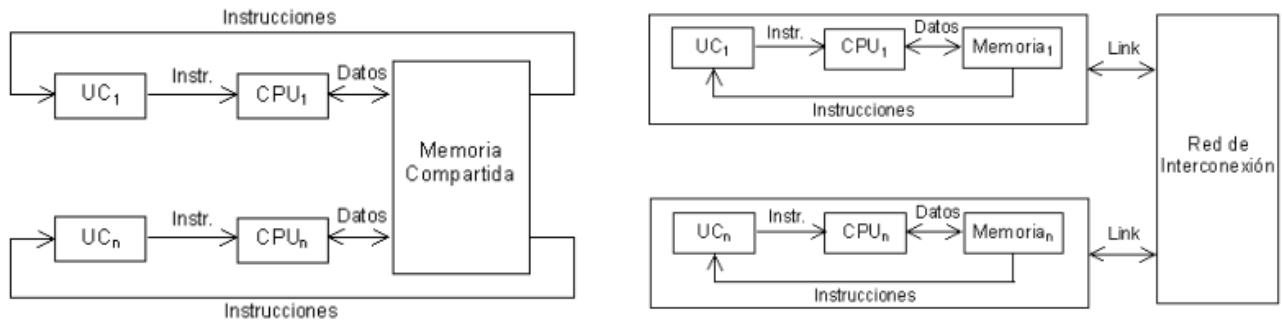
Multiple Instruction Multiple Data

Cada procesador tiene su propio flujo de instrucciones y de datos ⇒ cada uno ejecuta su propio “programa” a su ritmo

Pueden ser con memoria compartida o distribuida

### Sub-clasificación de MIMD:

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



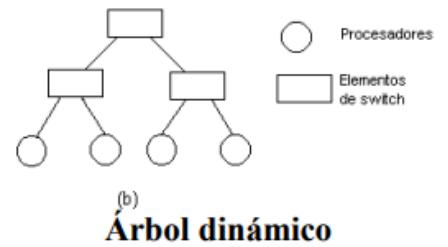
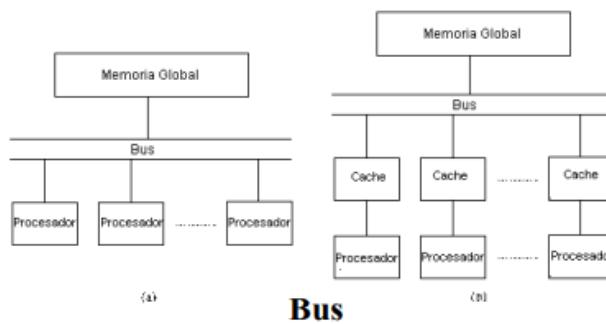
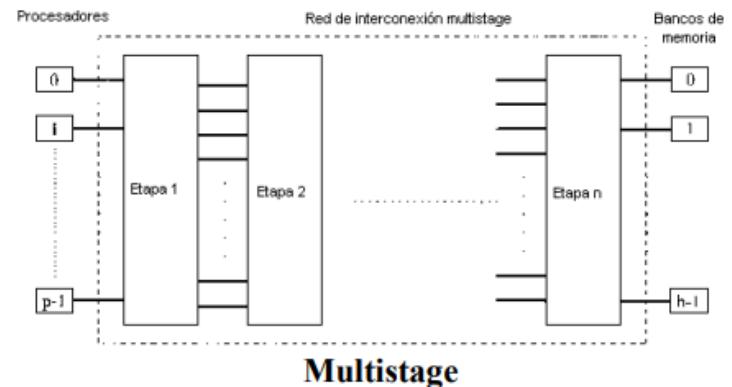
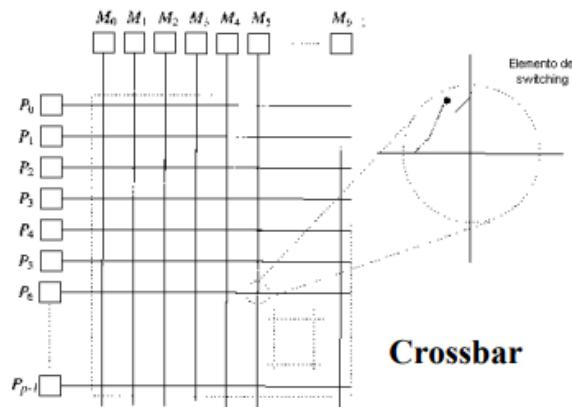
### Por la red de interconexión

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

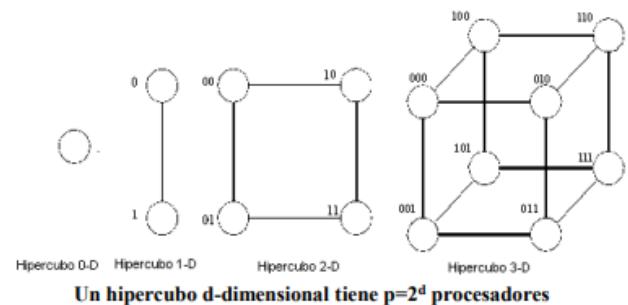
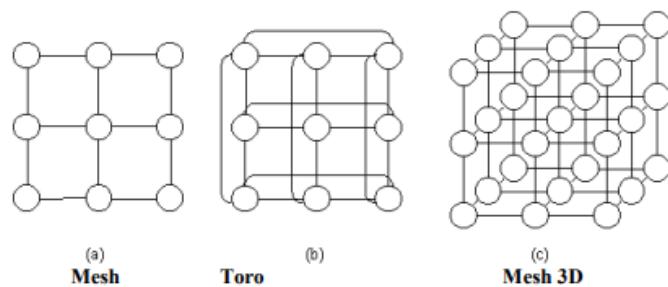
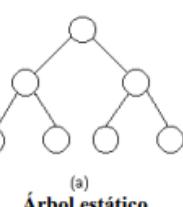
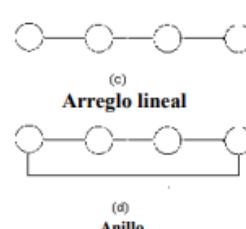
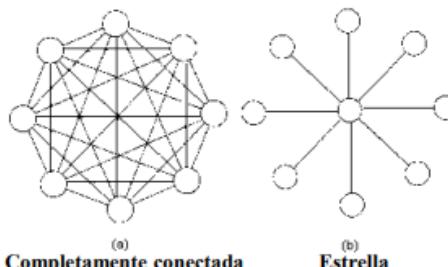
- Las redes estáticas:
  - Constan de links punto a punto
  - Típicamente se usan para máquinas de pasaje de mensajes
- Las redes dinámicas:
  - Están construidas usando switches y enlaces de comunicación
  - Normalmente para máquinas de memoria compartida

El diseño de la red de interconexión depende de una serie de factores (ancho de banda, tiempo de startup, paths estáticos o dinámicos, operación sincrónica o asincrónica, topología, costo, etc.)

## Redes de interconexión dinámicas



## Redes de interconexión estáticas



## Diseño de algoritmos paralelos

La mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes

Puede darse un enfoque metódico para maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas, y reducir el costo de backtracking por malas elecciones

Aspectos independientes de la máquina tales como la concurrencia son considerados tempranamente, y los aspectos específicos de la máquina se demoran

### ¿Por qué es compleja la programación paralela?

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

### Pasos a seguir

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

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

Pasos Fundamentales:

- Descomposición en Tareas

- Mapeo de Procesos a Procesadores

## Descomposición en tareas

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

Se trata de definir un gran número de pequeñas tareas para obtener una descomposición de grano fino, para brindar la mayor flexibilidad a los algoritmos paralelos potenciales

En etapas posteriores, la evaluación de los requerimientos de comunicación, arquitectura de destino, o temas de IS pueden llevar a descartar algunas posibilidades detectadas en esta etapa, revisando la partición original y aglomerando tareas para incrementar su tamaño o granularidad

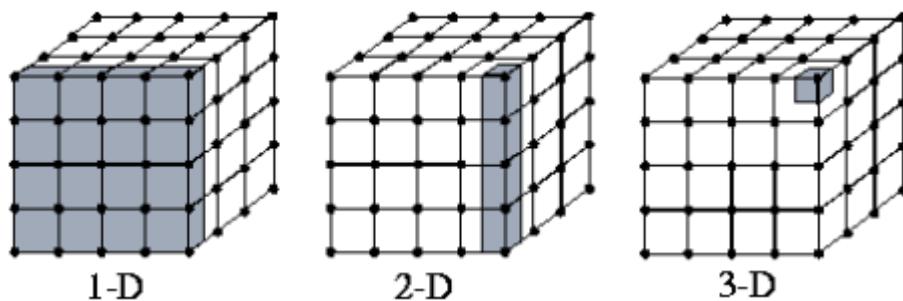
Esta descomposición puede realizarse de muchos modos. Un primer concepto es pensar en tareas de igual código (normalmente paralelismo de datos o dominio) pero también podemos tener diferente código (paralelismo funcional)

## Descomposición de datos

Determinar una división de los datos (en muchos casos, de igual tamaño) y luego asociarle el cómputo (típicamente, cada operación con los datos con que opera)

Esto da un número de tareas, donde cada uno comprende algunos datos y un conjunto de operaciones sobre ellos. Una operación puede requerir datos de varias tareas, y esto llevará a la comunicación

Son posibles distintas particiones, basadas en diferentes estructuras de datos. Por ejemplo, diferentes formas de descomponer una estructura 3D de datos. Inicialmente la de grano más fino



## Descomposición funcional

Primero descompone el cómputo en tareas disjuntas y luego trata los datos

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

Esta etapa revisa las decisiones tomadas con la visión de obtener un algoritmo que ejecute en forma eficiente en una clase de máquina real

En particular, se considera si es útil combinar o aglomerar las tareas para obtener otras de mayor tamaño. También se define si vale la pena replicar datos y/o computación

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

- Incremento de la granularidad: intenta reducir la cantidad de comunicaciones combinando varias tareas relacionadas en una sola
- Preservación de la flexibilidad: al juntar tareas puede limitarse la escalabilidad del algoritmo. La capacidad para crear un número variante de tareas es crítica si se busca un programa portable y escalable
- Reducción de costos de IS: 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 performance alcanzable por el algoritmo paralelo:

- Generación de las tareas (se conocen todas las tareas al inicio de la aplicación o se generan de manera dinámica)
- El tamaño de las tareas
- Conocimiento del tamaño de las tareas
- El volumen de datos asociado con cada tarea

## Mapeo de tareas a procesadores

Se especifica dónde ejecuta cada tarea

Este problema no existe en uniprocesadores o máquinas de memoria compartida con scheduling de tareas automático

Objetivo: minimizar tiempo de ejecución. Dos estrategias, que a veces conflictúan:

- Ubicar tareas que pueden ejecutar concurrentemente en  $\neq$  procesadores para mejorar la concurrencia
- 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

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

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

Notar que estos criterios pueden oponerse entre sí ... por ejemplo el criterio 3 puede llevarnos a NO paralelizar

Debe encontrarse un equilibrio que optimice el rendimiento paralelo  $\Rightarrow$  MAPPING DETERMINA LA EFICIENCIA DEL ALGORITMO

## Métricas de rendimiento

En el mundo serial la performance con frecuencia es medida teniendo en cuenta los requerimientos de tiempo y memoria de un programa

En un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance

Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución

A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores (sistema paralelo = algoritmo + arquitectura sobre la que se implementa)

La diversidad torna complejo el análisis de performance...

¿Qué interesa medir?

¿Qué indica que un sistema paralelo es mejor que otro?

¿Qué sucede si agrego procesadores?

En la medición de performance es usual elegir un problema y testear el tiempo variando el número de procesadores. Aquí subyacen las nociones de speedup y eficiencia, y la ley de Amdahl

Otro tema de interés es la escalabilidad, que da una medida de usar eficientemente un número creciente de procesadores

## Speedup (S)

S es el cociente entre el tiempo de ejecución del algoritmo serial conocido más rápido ( $T_s$ ) y el tiempo de ejecución paralelo del algoritmo elegido ( $T_p$  = tiempo entre que comienza a ejecutarse el primer proceso y termina el último)

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

El speedup óptimo es la mayor aceleración que podría encontrar en la arquitectura que estoy trabajando (si divido un problema en 4 tareas el speed up óptimo sería de 4 si los procesadores son homogéneos)

El speedup óptimo depende de la arquitectura cuando los procesadores son heterogéneos:

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

PotenciaCalculo(mejor) es la potencia de cálculo del mejor de los procesadores. El cociente de la fórmula entonces nos da la potencia relativa de cada una de las máquinas

Rango de valores: en general entre 0 y  $S_{óptimo}$

Speedup lineal o perfecto (speedup conseguido = speedup óptimo), sublineal (speedup conseguido < speedup óptimo) y superlineal (speedup conseguido > speedup óptimo)

El speedup superlineal se podría dar y es válido cuando para un algoritmo, al subdividir, se logra conseguir un mejor acceso a los datos al tenerlos en caché (al ser los conjuntos de datos más pequeños)

## Eficiencia

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

## Escalabilidad

Es muy difícil extrapolar la performance de un sistema paralelo, a partir de configuraciones con pocos procesadores y conjuntos de datos reducidos

No sirven los estudios con 2, 4, 8 procesadores que proyectan el Speedup alcanzable con 128 o 256 procesadores o el tiempo de procesamiento cuando tengamos 100 o 1000 veces más datos...

¿Por qué?

Básicamente porque los resultados con pequeños conjuntos de datos están afectados por la localidad en el manejo de la memoria, y los resultados con pocos procesadores porque las comunicaciones no computan los costos relacionados con la distancia entre procesadores y la disminución del ancho de banda efectivo

## Factores que limitan el speedup

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

## Paradigmas de programación paralela

Paradigma de programación: clase de algoritmos que resuelve distintos problemas, pero tienen la misma estructura de control

Para cada paradigma puede escribirse un esqueleto algorítmico que define la estructura de control común

Dentro de la programación paralela pueden encontrarse paradigmas que permiten encuadrar los problemas en alguno de ellos

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

## Cliente/servidor

Cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido

Los servidores son procesos que esperan pedidos de servicios de múltiples clientes

Naturalmente unos y otros pueden ejecutarse en procesadores diferentes

Comunicación bidireccional

Atención de a un cliente a la vez, o a varios con multithreading

Mecanismos de invocación variados (rendezvous, RPC, monitores)

El soporte distribuido puede ser simple (LAN) o extendido a la WEB

## Master/slave o master/worker

Basado en organizaciones del mundo real

El master envía iterativamente datos a los workers y recibe resultados de éstos

Possible “cuello de botella” (por ejemplo, por tareas muy chicas o slaves muy rápidos)

→ elección del grano adecuado

Dos casos de acuerdo a las dependencias de las iteraciones:

- Iteraciones dependientes: el master necesita los resultados de todos los workers para generar un nuevo conjunto de datos
- Entradas de datos independientes: los datos llegan al maestro, que no necesita resultados anteriores para generar un nuevo conjunto de datos

Dos opciones para la distribución de los datos:

- Distribuir todos los disponibles, de acuerdo a alguna política (estático)
- Bajo petición o demanda (dinámico)

Existen variantes, pero básicamente un procesador es responsable de la coordinación y los otros de resolver los problemas asignados

Es una variación de SPMD donde hay dos programas en lugar de sólo uno

Casos:

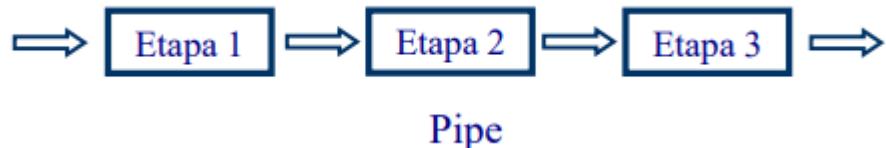
- Procesadores heterogéneos y con distintas velocidades → problemas con el balance de carga
- Trabajo que debe realizarse en “fases” → sincronización
- Generalización a modelo multi-nivel o jerárquico

## Pipeline y Algoritmos Sistólicos

El problema se partitiona en una secuencia de pasos. El stream de datos pasa entre los procesos, y cada uno realiza una tarea sobre él

Ejemplo: filtrado, etiquetado y análisis de escena en imágenes

Mapeo natural a un arreglo lineal de procesadores



Extensiones:

- Procesadores especializados no iguales
- Más de un procesador para una tarea determinada
- El flujo puede no ser una línea simple (ejemplo: ensamble de autos con varias líneas que son combinadas) → procesamiento sistólico

## Dividir y conquistar

En general implica paralelismo recursivo donde el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (dividir y conquistar)

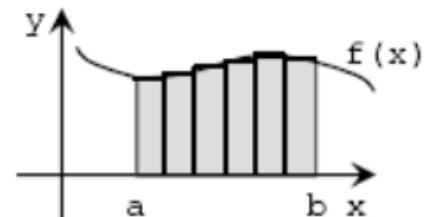
División repetida de problemas y datos en subproblemas más chicos (fase de dividir); resolución independiente de éstos (conquistar), con frecuencia de manera recursiva. Las soluciones son combinadas en la solución global (fase de combinar)

La subdivisión puede corresponderse con la descomposición entre procesadores. Cada subproblema puede mapearse a un procesador. Cada proceso recibe una fracción de datos: si puede los procesa; sino, crea un n° de "hijos" y les distribuye los datos

Ejemplos clásicos son el "sorting by merging", el cálculo de raíces en funciones continuas, problema del viajante

## Problema de la cuadratura

Calcular una aproximación de la integral de una función continua  $f(x)$  en el intervalo de  $a$  a  $b$

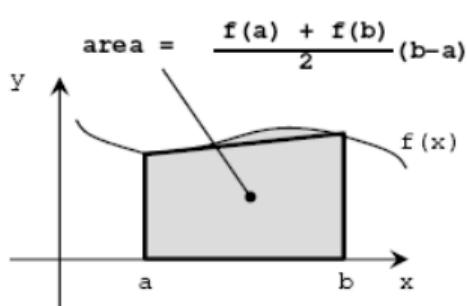


Solución secuencial iterativa (usando el método trapezoidal):

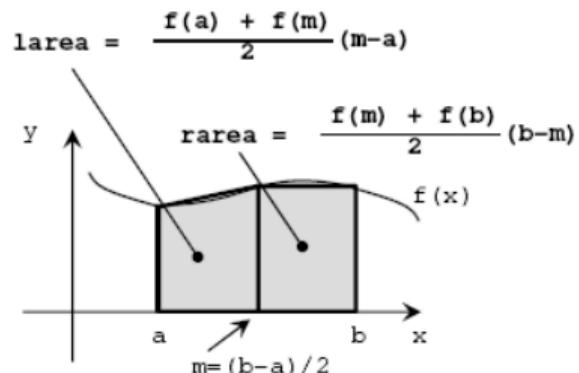
```

double fl = f(a), fr, area = 0.0;
double dx = (b-a)/ni;
for [x = (a+dx) to b by dx]
{
    fr = f(x);
    area = area + (fl+fr) * dx / 2;
    fl = fr;
}

```



(a) First approximation (area)



(b) Second approximation  
(larea + rarea)

Si  $\text{abs}((\text{larea} + \text{rarea}) - \text{area}) > \epsilon$ , repetir el cálculo para cada intervalo  $[a,m]$  y  $[m,b]$  de manera similar hasta que la diferencia entre aproximaciones consecutivas esté dentro de un dado  $\epsilon$

## Procedimiento secuencial

```
double quad(double l,r,f1,fr,area) {  
    double m = (l+r)/2;  
    double fm = f(m);  
    double larea = (f1+fm)*(m-l)/2;  
    double rarea = (fm+fr)*(r-m)/2;  
    if (abs((larea+rarea)-area) > e) {  
        larea = quad(l,m,f1,fm,larea);  
        rarea = quad(m,r,fm,fr,rarea);  
    }  
    return (larea+rarea);  
}
```

## Procedimiento paralelo

```
double quad(double l,r,f1,fr,area) {  
    double m = (l+r)/2;  
    double fm = f(m);  
    double larea = (f1+fm)*(m-l)/2;  
    double rarea = (fm+fr)*(r-m)/2;  
    if (abs((larea+rarea)-area) > e) {  
        co larea = quad(l,m,f1,fm,larea);  
        || rarea = quad(m,r,fm,fr,rarea);  
        oc  
    }  
    return (larea+rarea);  
}
```

Dos llamados recursivos son independientes y pueden ejecutarse en paralelo

## SPMD

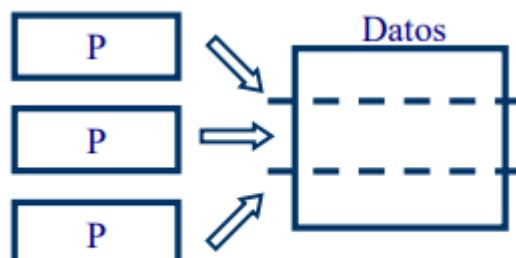
El programador genera un programa único que ejecuta cada nodo sobre una porción del dominio de datos

La diferente evaluación de un predicado en sentencias condicionales permite que cada nodo tome distintos caminos del programa

Dos fases:

1. Elección de la distribución de datos
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



SPMD

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

Multiplicación de matrices

### Solución secuencial:

```
double a[n,n], b[n,n], c[n,n];
for [i = 1 to n]
  { for [j = 1 to n]
    { c[i,j] = 0;
      for [k = 1 to n]
        c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
    }
  }
```

$$\begin{array}{l}
 \left[ \begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array} \right] \times \left[ \begin{array}{cccc} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{array} \right] = \left[ \begin{array}{cccc} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{array} \right] \\
 \left[ \begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array} \right] \times \left[ \begin{array}{cccc} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{array} \right] = \left[ \begin{array}{cccc} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{array} \right] \\
 \cdots \\
 \left[ \begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array} \right] \times \left[ \begin{array}{cccc} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{array} \right] = \left[ \begin{array}{cccc} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{array} \right]
 \end{array}$$

El loop interno calcula el producto interno de la fila i de la matriz a por la columna j de la matriz b y obtiene  $c[i,j]$

El cómputo de cada producto interno es independiente. Aplicación embarrassingly parallel (muchas operaciones en paralelas)

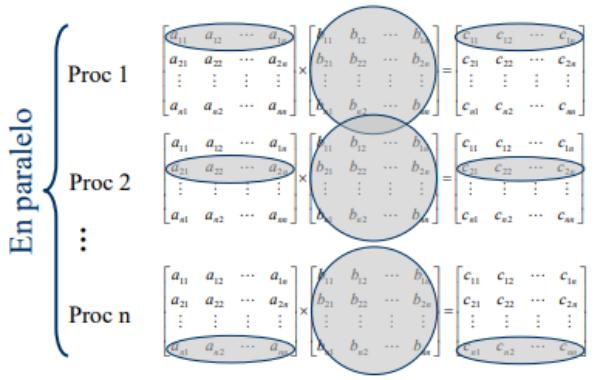
Diferentes acciones paralelas posibles

## Solución paralela por fila:

```

double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { for [j = 1 to n]
    { c[i,j] = 0;
      for [k = 1 to n]
        c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
    }
  }
}

```

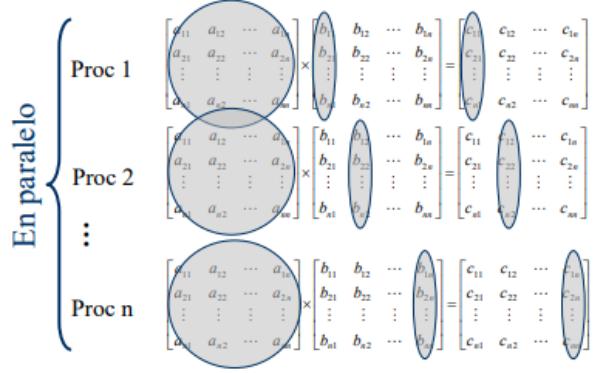


## Solución paralela por columna:

```

double a[n,n], b[n,n], c[n,n];
co [j = 1 to n]
{ for [i = 1 to n]
  { c[i,j] = 0;
    for [k = 1 to n]
      c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
  }
}

```

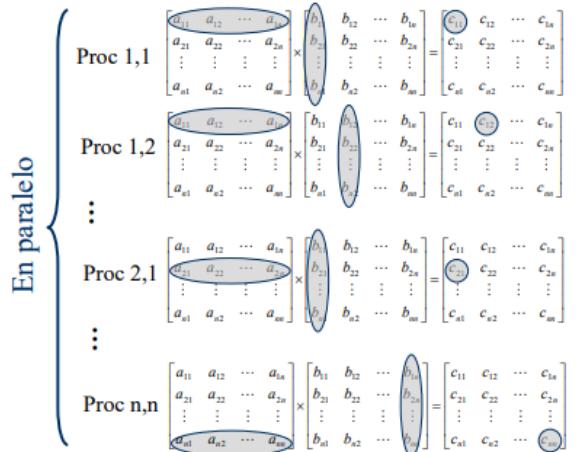


## Solución paralela por celda (opción 1):

```

double a[n,n], b[n,n], c[n,n];
co [i = 1 to n , j= 1 to n]
  { c[i,j] = 0;
    for [k = 1 to n]
      c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
  }

```



## Solución paralela por celda (opción 2):

```

double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
{ co [j = 1 to n]
  { c[i,j] = 0;
    for [k = 1 to n]
      c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
  }
}

```

## Solución paralela por fila con process:

```

process fila [i = 1 to n]
{ for [j = 1 to n]
  { c[i,j] = 0;
    for [k = 1 to n]
      c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
  }
}

```

¿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

## Solución paralela por strips:

(P procesadores con P<n)

```

process worker [ w = 1 to P]
{ int primera = (w-1)*(n/P) + 1;
  int ultima = primera + (n/P) - 1;
  for [i = primera to ultima]
    { for [j = 1 to n]
      { c[i,j] = 0;
        for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
    }
}

```

En paralelo

$$\begin{array}{c}
 \text{Worker 1} \\
 \vdots \\
 \text{Worker P}
 \end{array}
 \left[ \begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{s1} & a_{s2} & \cdots & a_{sn} \end{array} \right] \times \left[ \begin{array}{cccc} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{t1} & b_{t2} & \cdots & b_{tn} \end{array} \right] = \left[ \begin{array}{cccc} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{u1} & c_{u2} & \cdots & c_{un} \end{array} \right]$$

n/P filas

$$\left[ \begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{s1} & a_{s2} & \cdots & a_{sn} \end{array} \right] \times \left[ \begin{array}{cccc} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{t1} & b_{t2} & \cdots & b_{tn} \end{array} \right] = \left[ \begin{array}{cccc} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{u1} & c_{u2} & \cdots & c_{un} \end{array} \right]$$

n/P filas