Resumen Concurrente

La concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente. Permite a distintos objetos actuar al mismo tiempo. Es un concepto clave dentro de la computación; relevante para el diseño de hard, SO multiprocesadores, computación distribuida, programación y diseño. La necesidad de sistemas de cómputo cada vez más poderosos y flexibles atenta contra la simplificación de asunciones de secuencialidad. La concurrencia está presente en la naturaleza (sistemas biológicos que comprenden un gran número de células evolucionando simultáneamente y realizando procesos independientes), la vida diaria, los sistemas de cómputo, etc. Cualquier sistema más o menos "inteligente" o complejo exhibe concurrencia. Desde un smartphone hasta un automóvil; navegador web accediendo a información mientras atiende al usuario; acceso de varias aplicaciones a disco, varios usuarios conectados al mismo sistema (ej. haciendo una reserva), juegos.

Como no hay más ciclos de reloj, ahora hay multicores, por lo que es necesaria la programación concurrente. Esto permite aplicaciones con estructura más natural, ya que el mundo no es secuencial, y es más apropiado programar múltiples actividades interdependientes y concurrentes, con reacción a entradas asincrónicas como sensores. Además, se obtiene una mejora en la respuesta, ya que no se bloquea la aplicación completa por E/S y se incrementa el rendimiento de la aplicación por mejor uso del hardware (ejecución paralela). La concurrencia también da lugar a sistemas distribuidos: una aplicación en varias máquinas, sistemas C/S o P2P.

Objetivos de los sistemas concurrentes: el mundo real es concurrente, por lo que se busca 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 procesamiento de datos, 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, y explotación de la concurrencia inherente a la mayoría de los problemas reales.

Concurrencia a nivel de hardware: hay un límite físico en la velocidad de los procesadores. Las máquinas monoprocesador ya no pueden mejorar, por lo que se optó por más procesadores por chip para tener mayor potencia de cómputo, pero para usarlos eficientemente se necesita programación concurrente y paralela.
Multiprocesadores de memoria compartida: la interacción se da modificando datos almacenados en la MC. Esquemas UMA (uniform memory access, memoria compartida entre todos los procesadores, problemas de sincronización y consistencia) y NUMA (cada procesador tiene su memoria local, problemas de consistencia).
Multiprocesadores con memoria distribuida: procesadores conectados por una red, c/u con su memoria local y la interacción es sólo por pasaje de mensajes.

Programa secuencial: un solo flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente. Un único thread o flujo de control significa un programa secuencial monoprocesador. Múltiples threads o flujos de control significan programa concurrente, procesos paralelos.

Proceso: programa secuencial.

Posibles comportamientos de procesos

Procesos independientes: relativamente raros y poco interesantes.
Competencia: típico en SO y redes debido a recursos compartidos. Conceptos de deadlock y starvation: starvation deniega siempre el acceso a un recurso compartido, sin el cual la tarea no puede finalizar. Deadlock produce inanición, pero la inanición puede terminar sola, mientras que el deadlock necesita una acción del exterior.
Cooperación: los procesos se combinan para resolver una tarea común. Requieren sincronización.

Ejemplo de procesamiento secuencial, concurrente y paralelo: fabricar un objeto compuesto por N partes.

Secuencial y monoprocesador (una máquina): la solución secuencial nos fuerza a establecer un estricto orden temporal. Al disponer de sólo una máquina el ensamblado final del objeto sólo se podrá realizar luego de N pasos de procesamiento o fabricación.
Paralela (multiplico hard): si disponemos de N máquinas para fabricar el objeto, y no hay dependencias (como materia prima), cada una puede trabajar al mismo tiempo en una parte. 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, esperarse en puntos clave, comunicación, tratamiento ante fallas, asignación de una máquina para ensamblado.
Otro enfoque: una máquina dedica parte del tiempo a cada componente del objeto → concurrencia sin paralelismo de hard. Dificultades: distribución de carga de trabajo, compartir recursos, esperarse en puntos claves, comunicación, recuperar el “estado” de cada proceso al retormarlo. En la multiprogramación en un procesador, el tiempo de CPU es compartido entre varios procesos, ej por time slicing. El SO controla y planifica procesos: si el slice expiró o el proceso de bloquea, el SO hace context switch (suspender el proceso actual y restaurar otro).
Concurrencia: concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.

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 (en paralelo), si el hardware lo permite. Un programa concurrente puede tener N procesos habilitados para ejecutarse concurrentemente y un sistema concurrente puede disponer de M procesadores cada uno de los cuales puede ejecutar uno o más procesos.
Características importantes: interacción, no determinismo (dificultad para interpretación y debug), ejecución infinita.

Concurrencia vs. paralelismo: la concurrencia no es (sólo) paralelismo.
Concurrencia “interleaved”: procesamiento simultáneo lógicamente, ejecución intercalada en un único procesador. “Pseudo-paralelismo”.
Concurrencia simultánea: procesamiento simultáneo físicamente, requiere un sistema multiprocesador. Paralelismo “full”.

Procesos e hilos: todos los SO soportan procesos. Cada proceso se ejecuta en una máquina virtual distinta. Algunos SO soportan procesos ligeros (hilos o threads), que son procesos livianos que tienen su propio contador de programa y su pila de ejecución, pero no controlan el "contexto pesado" (por ej. la tabla de páginas). Todos los hilos de un proceso comparten la misma máquina virtual y tienen acceso al mismo espacio de memoria. El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias. La concurrencia puede estar soportada por el lenguaje (Java, Ada, occam2) o el SO (C/POSIX).

Secuencialidad y concurrencia
Un programa secuencial está totalmente ordenado y es determinístico: para los mismos datos de entrada, ejecuta siempre la misma secuencia de instrucciones y obtiene la misma salida, pero esto no es verdad para los programas concurrentes. Ej:
x=0; //P
y=0; //Q
z=0; //R
en este caso, el orden de ejecución es irrelevante, tener la computación distribuida en 3 máquinas sería más rápido. Son instrucciones que pueden ser lógicamente concurrentes. Nota: hay instrucciones que requieren ejecución secuencial, como x=1; x=x+1;
Orden parcial: las instrucciones pueden tener overlapping: si descomponemos P en p1, p2, .., pn (también a Q y R) podemos tener los ordenamientos: p1, p2, q1, r1, q2 …
La única regla es que pi ejecuta antes de pj si i<j (orden parcial).
En el caso anterior, cualquiera de las ejecuciones con distinto orden dará el mismo resultado, pero por lo general, diferentes ejecuciones con la misma entrada pueden dar distintos resultados:
suponer que x es 5
x=0; //P
x=x+1; //Q

Escenario 1: P → Q da x=1
Escenario 2: Q → P da x=0

Los programas concurrentes pueden ser no-determinísticos: dar distintos resultados al ejecutarse sobre los mismos datos de entrada.

<u>Clases de aplicaciones</u>

Sistemas Multithreaded: (multiprogramación) los procesos comparten uno o más procesadores. Ejecución de N procesos independientes en M procesadores (N>M). Un sistema de software de "multithreading" maneja simultáneamente tareas independientes, asignando los procesadores de acuerdo a alguna política (ej. por tiempos). Organización más "natural" como un programa concurrente. Ej: sistemas de ventanas, SO time-shared y multiprocesador, sistemas de tiempo real.
Sistema de cómputo paralelo: (multiprocesamiento) cada proceso corre en su propio procesador pero con memoria compartida. Resolver un problema en el menor tiempo o un problema + grande en aprox. el mismo tiempo usando una arq. multiprocesador en la que se pueda distribuir la tarea global en tareas que puedan ejecutarse en distintos procesadores. Paralelismo de datos y paralelismo de procesos. Ej: cálculo científico, procesamiento de imágenes. Ej simple: suma de 32

números, 1 persona (secuencial) 32-1 sumas c/u 1 unidad de tiempo, 2 personas 16 sumas, la mitad del tiempo, máxima concurrencia 16 personas formando un árbol binario, el resultado lo obtiene el último sumador. El procesamiento paralelo lleva a los conceptos de speedup y eficiencia.
Sistemas de cómputo distribuido: cada proceso corre en su propio procesador conectado a los otros a través de una red. Una red de comunicaciones vincula procesadores diferentes sobre los que se ejecutan procesos que se comunican esencialmente por mensajes. Cada componente del sist. distribuido puede hacer a su vez multithreading. Ej: servidores de archivos en una red, sistemas de BD en bancos y aerolíneas (acceso a datos remotos).

Conceptos básicos de concurrencia

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

- **Memoria compartida:** los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella. Lógicamente no pueden operar simultáneamente sobre la MC, lo que obliga a bloquear y liberar el acceso a la memoria (ej. semáforos).
- **Pasaje de mensajes:** es necesario establecer un canal (lógico o físico) para transmitir información entre procesos. También el lenguaje debe proveer un protocolo adecuado.

Los procesos se **sincronizan** por exclusión mutua en el acceso a secciones críticas de código para no ejecutar simultáneamente, y por condición. Ej: completar escrituras antes de que comience una lectura, que un cajero de dinero luego de verificar la tarjeta.
Sincronizar: posesión de información acerca de otro proceso para coordinar actividades.
Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más *acciones atómicas* (indivisibles, los estados intermedios son invisibles para los otros procesos).
La *historia* (trace) de un programa concurrente es una ejecución dada por un intercalado (interleaving) particular de acciones individuales de los procesos. S0→S1→...→SN. El número posible de historias de un programa concurrente es generalmente enorme. Un programa concurrente con n procesos, donde c/u ejecuta m acciones atómicas tiene una cantidad de historias dada por $(n*m)!/(m!)^n$. Para 3 procesos con 2 acciones hay 90 interleavings posibles.
Algunas historias son válidas y otras no. El objetivo de la sincronización es restringir las historias de un programa concurrente sólo a las permitidas.

Formas de sincronización

Sincronización por exclusión mutua: asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo. Si el programa tiene secciones críticas que pueden compartir más de un proceso, EM evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.
Sincronización por condición: permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.
Ej de los dos mecanismos de sincronización en un problema de utilización de un área de memoria compartida: buffer limitado con productores y consumidores.

Otros conceptos

Prioridad: un proceso que tiene mayor prioridad puede causar la suspensión (pre-emption) de otro proceso o tomar un recurso compartido obligando a otro proceso a retirarse.

Elección de granularidad: optimizar la relación entre el número de procesadores y el tamaño de memoria total.

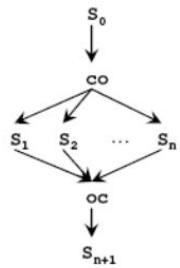
Manejo de los recursos: uno de los temas principales de la programación concurrente es la administración de recursos compartidos, que incluye la asignación de recursos, métodos de acceso a recursos, bloqueo y liberación de recursos, seguridad y consistencia. Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (fairness). Dos situaciones NO deseadas son la **inanición** de un proceso (no logra acceder a los recursos) 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:

- Recursos reusables serialmente: los procesos comparten recursos que pueden usar con EM.
- 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 c/u tiene un recurso que su sucesor en el ciclo está esperando adquirir.

Requerimientos para un lenguaje concurrente: independientemente del mecanismo de comunicación/sincronización entre procesos, los lenguajes de programación concurrente deberán proveer primitivas adecuadas para la especificación e implementación de las mismas. Se requiere:



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

Resumen de conceptos: la concurrencia es un concepto de software (propiedad del programa). La programación paralela se asocia con la ejecución concurrente en múltiples procesadores que pueden tener memoria compartida, y con un objetivo de incrementar performance (propiedad de la máquina). La programación distribuida es un “caso” de concurrencia con múltiples procesadores y sin memoria compartida.

En programación concurrente la organización de procesos y procesadores constituye la arquitectura del sistema concurrente. Especificar la concurrencia es esencialmente especificar los procesos concurrentes, su comunicación y sincronización.

Clase 2

Concurrencia

- Sentencia co: co S_1 // … // S_n oc → Ejecuta las S_i tareas concurrentemente. La ejecución del co termina cuando todas las tareas terminaron.
Cuantificadores:
co [i=1 to n] { a[i] = 0; b[i] = 0 } oc → Crea n tareas concurrentes

- Process: otra forma de representar concurrencia.
process A {sentencias} → proceso único independiente
Cuantificadores:
process B [i=1 to n] {sentencias} → n procesos independientes.

Diferencias: process ejecuta en background, mientras el código que contiene un co espera a que el proceso creado por la sentencia co termine antes de ejecutar la siguiente sentencia.

Ejemplo: qué imprime en cada caso? no determinismo...

```
process imprime10{
        for [i=1 to 10]
                write (i);}
```

```
process imprime1[i=1 to 10]{
        write (i);
}
```

Paradigmas de resolución de programas concurrentes

1) Paralelismo iterativo: un programa consta de un conjunto de procesos (posiblemente idénticos) c/u de los cuales tiene 1 o más loops; cada proceso es un programa iterativo. Los procesos cooperan para resolver un único problema (por ej un sist. de ecuaciones), pueden trabajar independientemente, y comunicarse y sincronizar por memoria compartida o MP. Generalmente, el dominio de los datos se divide entre los procesos siguiendo diferentes patrones.
2) Paralelismo recursivo: el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (Dividir y conquistar). Ej. sorting by merging.
3) Productores y consumidores: muestran procesos que se comunican. Es habitual que estos procesos se organicen en pipes a través de los cuales fluye la información. Cada proceso en el pipe es un filtro que consume la salida de su proceso predecesor y produce una salida para el proceso siguiente. Ej. a distintos niveles de SO.
4) Cliente-servidor: es el esquema dominante en las aplicaciones de procesamiento distribuido. Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Unos y otros pueden ejecutarse en procesadores diferentes. Comunicación bidireccional. Atención de a un cliente o con multithreading a varios. Mecanismos de invocación variados (por ej. rendezvous y RPC en MD, monitores en MC). El soporte distribuido puede ser simple (LAN) o extendido a la web.
5) Pares que interactúan: los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea y completar el objetivo. Permite mayor grado de asincronismo que C/S.

Configuraciones posibles: grilla, pipe circular, uno a uno, arbitraria.

Paralelismo iterativo: 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] {
                # computa el producto interno de a[i,*] y b[*,j]
                c[i,j] = 0.0;
                for [k = 1 to n]
                        c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
}
```

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, 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 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]);
                }
        }
}
```

Volviendo al hardware: se puede identificar diferentes enfoques para clasificar las arquitecturas paralelas.

- Por la organización del espacio de direcciones:

-Multiprocesadores de memoria compartida: interacción modificando datos en la memoria compartida. Problema de consistencia. UMA y NUMA.

-Multiprocesadores con memoria distribuida: memoria local (no hay problemas de consistencia). Interacción sólo por pasaje de mensajes.

- Por mecanismos de control:

- SISD (Single Instruction Single Data): instrucciones ejecutadas en secuencia, una por ciclo de ejecución. La memoria afectada es usada sólo por esta instrucción. Usada por la mayoría de los uniprocesadores.
- MISD: los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes.
- SIMD: conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos.
- MIMD: cada procesador tiene su propio flujo de instrucciones y de datos, c/u ejecuta su propio programa. Puede ser con memoria compartida o distribuida.

- Por granularidad de los procesadores: la granularidad es la relación entre el n° de procesadores y el tamaño de memoria total.

- Grano grueso (coarse-grained): pocos procesadores muy poderosos.
- Grano fino: gran número de procesadores menos potentes.
- Grano medio.

Si las aplicaciones tienen concurrencia limitada, pueden usar eficientemente pocos procesadores, por lo que conviene máquinas de grano grueso. Las máquinas de grano fino son más efectivas en costo para aplicaciones con alta concurrencia.

- Por red de interconexión: tanto las máquinas de MC como de MP pueden construirse conectando procesadores y memorias usando diversas redes de interconexión:

- estáticas: constan de links punto a punto, típicamente se usan para máquinas de MP.

- dinámicas: construidas usando switches y enlaces de comunicación. Normalmente para máquinas de MC.

Acciones Atómicas y Sincronización

Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas. Una acción atómica hace una transformación de estado indivisibles (estados intermedios invisibles para otros procesos). La ejecución de un programa concurrente es un interleaving de las acciones atómicas ejecutadas por procesos individuales. No todos los interleavings son aceptables.

Sincronizar:

- Combinar acciones atómicas de grano fino en acciones (compuestas) de grano grueso que den la exclusión mutua.
- Demorar un proceso hasta que el estado de programa satisfaga algún predicado (por condición).

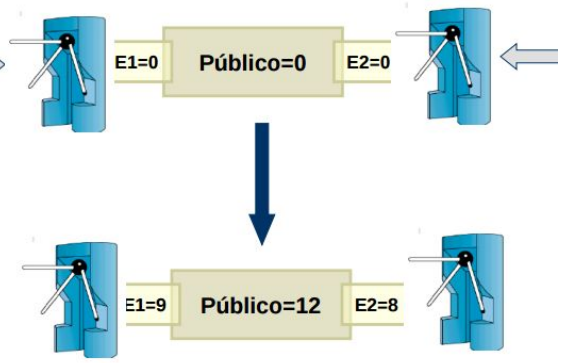
El objetivo de la sincronización es prevenir los interleavings indeseables restringiendo las historias de un programa concurrente sólo a las permitidas.
Una acción atómica de grano fino se debe implementar por hardware. La operación A=A+3 no es atómica, ya que es necesario cargar el valor de A en un registro, sumarle 3 y volver a cargar el valor del registro en A.

El problema de la interferencia: cuando un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

Ejemplo: ¿Qué puede suceder con los valores de E1, E2 y público?

```
process 1
 { while (true)
      esperar llegada
      E1 = E1 + 1;
      Público = Público + 1;
 }

process 2
 { while (true)
      esperar llegada
      E2 = E2 + 1;
      Público = Público + 1;
 }
```



Atomicidad de grano fino

```
y = 4; x = 0; z=2;
co
        x = y + z (1) → 1.1 load, 1.2 add, 1.3 store
        // y = 3 (2) → store
        // z = 4 (3) → store
oc
```

y = 3, z = 4 en todos los casos

x puede ser:
6 si ejecuta (1)(2)(3) o (1)(3)(2)
5 si ejecuta (2)(1)(3)
6 si ejecuta (3)(1)(2)
7 si ejecuta (2)(3)(1) o (3)(2)(1)
6 si ejecuta (1.1)(2)(1.2)(1.3)(3)
8 si ejecuta (1.1)(3)(1.2)(1.3)(2)
…

Otro ejemplo:

```
x = 2; y = 2;
co
z = x + y (1)
// x = 3; y = 4 (2)
oc
```

x=3, y=4 en todos los casos. z puede ser 4, 5, 6 o 7, pero nunca podríamos parar el programa y ver un estado en que x+y=6.

Si una expresión **e** en un proceso no referencia a 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. Pero normalmente los programas concurrentes no son disjuntos.

Propiedad de "A lo sumo una vez"

Una referencia crítica en una expresión es una referencia a una vble que es modificada por otro proceso. Se asume que toda referencia crítica es a una vble simple leída y escrita atómicamente.
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.

Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez. Una definición similar se aplica a expresiones que no están en sentencias de asignación (satisface ASV si no contiene más de una ref. crítica).

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

```
int x=0, y=0;
co x=x+1 // y=y+1 oc;          No hay ref. críticas en ningún proceso

int x=0, y=0;                  El 1er proc tiene 1 ref. crítica. El 2do ninguna
```

co x=y+1 // y=y+1 oc; y=1 siempre, x=1 o 2

int x=0, y=0; Ninguna asignación satisface ASV.
co x=y+1 // y=x+1 oc; Posibles: x=1, y=2 / x=2, y=1 / x=1, y=1

Especificación de la sincronización: si una expresión o asignación no satisface ASV, con frecuencia es necesario ejecutarla atómicamente. En general, es necesario ejecutar secuencias de sentencias como una única acción atómica. Una acción atómica de grano grueso es una especificación en alto nivel del comportamiento requerido de un programa que puede ser implementada de distintas maneras, dependiendo del mecanismo de sincronización disponible.
<e> indica que la expresión **e** debe ser evaluada atómicamente.
<await (B) S;> se utiliza para especificar sincronización. La expresión booleana B especifica una condición de demora. 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.

Sólo exclusión mutua: <S>. Ej: <x=x+1; y=y+1>
Sólo sincronización por condición: <await (B)>. Ej: <await (count>0)>. Si B satisface ASV, puede implementarse como busy waiting o spinning:
do (not B) → skip od // (while (not B);)

Ejemplo: productor/consumidor con buffer de tamaño N.

```
cant: int = 0;
Buffer: cola;

process Productor{
        while (true)
                <await (cant < N); push(buffer, elemento); cant++ >
}
process Consumidor{
        while (true)
                <await (cant > 0); pop(buffer, elemento); cant-- >
}
```

<u>Seguridad y Vida</u>

Seguridad: nada malo le ocurre a un objeto. Asegura estados consistentes. Una falla de seguridad indica que algo anda mal. Ej. ausencia de deadlock y ausencia de interferencia (exclusión mutua) entre procesos.
Vida: eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks. Una falla de vida indica que se deja de ejecutar. Ej. terminación, asegurar que un pedido de servicio será atendido, que un mensaje llegue a destino, que un proceso eventualmente alcanzará su SC, etc. Depende de las políticas de scheduling.

Ejemplo: puente de una sola vía, los autos pueden moverse concurrentemente si van en la misma

dirección. Violación de seguridad: que dos autos en distintas direcciones entren al puente al mismo tiempo. Vida: cada auto tendrá eventualmente la oportunidad de cruzar el puente?

Fallas típicas

Seguridad

- Conflictos de read/write: un proceso lee un campo y otro lo escribe (el mismo valor visto por el lector depende de quién ganó).
- Conflictos de write/write: dos procesos escriben el mismo campo.

Vida

- Temporarias: bloqueos temporarios, espera, contención de CPU, falla recuperable.
- Permanente: deadlock, señales perdidas, anidamiento de bloqueos, inanición.

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 ejecutado. Si hay varios procesos, hay varias acciones atómicas elegibles; una política de scheduling determina cuál será la próxima en ejecutarse.
Qué sucede en este caso?

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

Fairness incondicional: una política de scheduling es incondicionalmente fair (o imparcial) si toda acción atómica incondicional que es elegible eventualmente es ejecutada.
Fairness débil: una política de scheduling es débilmente fair si (1) es incondicionalmente fair y (2) toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece tru hasta que es vista por el proceso que ejecuta la acción atómica condicional. No es suficiente para asegurar que cualquier sentencia await elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de false a true y nuevamente a false) mientras un proceso está demorado.
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.

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 ej anterior, con 1 procesador, una política que alterna las acciones de los procesos sería fuertemente fair, pero es impráctica. RR es práctica pero no es fuertemente fair.

Clase 3

Sincronización por variables compartidas: Locks y barreras

En la técnica de busy waiting, un proceso chequea repetidamente una condición hasta que sea verdadera. Ventaja: implementación con instrucciones de cualquier procesador. Ineficiente en multiprogramación (cuando varios procesos comparten el procesador y la ejecución es interleaved). Aceptable si cada proceso ejecuta en su procesador.

El problema de la sección crítica

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

Los protocolos de entrada y de salida deben satisfacer las siguientes propiedades:

- **Exclusión Mutua:** a lo sumo un proceso está en su SC
- **Ausencia de deadlock:** si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.
- **Ausencia de demora innecesaria:** si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.
- **Eventual entrada:** un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

Las 3 primeras son propiedades de seguridad, y la 4° de vida.
Solución trivial <SC>, pero cómo implementar los <>?

Solución de "grano grueso"

```
bool in1=false, in2=false
{MUTEX: ¬(in1 ∧ in2)}
process SC1 {
        while (true) { ⟨await (not in2) in1 = true; ⟩      #protocolo de entrada
        sección crítica;
        in1 = false;                                        #protocolo de salida
        sección no crítica;
        }
```

```
}

process SC2 {
        while (true) { ⟨await (not in1) in2 = true; ⟩      #protocolo de entrada
        sección crítica;
        in2 = false;                                       #protocolo de salida
        sección no crítica;
        }
}
```

Satisface las 4 condiciones?
Exclusión mutua: por construcción, p1 y p2 se excluyen en el acceso a la SC.
Ausencia de deadlock: si hay deadlock, p1 y p2 están bloqueados en su protocolo de entrada; 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 la SC, cada proceso vuelve a serlo).
Ausencia de demora innecesaria: si p1 está fuera de su SC o terminó, in1 es false. Si p2 está tratando de entrar a SC y no puede, la guarda en su protocolo de entrada debe ser falsa (in1 en true). ¬in1 ^ in1 = false → no hay demora innecesaria.
Eventual entrada: si p1 está tratando de entrar a su SC y no puede, p2 está en su SC e in2 es true. Un proceso que está en SC eventualmente sale; in2 será false y la guarda de p1 verdadera. Análogamente para p2. Si los procesos corren en procesadores iguales y el tiempo de acceso a SC es finito, las guardas son true con infinita frecuencia. Luego, se garantiza la eventual entrada con una política de scheduling fuertemente fair (aunque estas políticas son imprácticas en la mayor parte de los casos).

Solución de “grano fino”

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.
TS toma una variable compartida **lock** como argumento y devuelve un resultado booleano:

```
bool TS(bool lock){
        <bool initial = lock;     # guarda valor inicial
        lock=true;                # setea lock
        return initial;>          # devuelve valor inicial
}
```

Spin locks

```
bool lock=false;          # lock compartido
process SC[i=1 to n] {
        while (true) {
                while (TS(lock)) skip ;          # protocolo de entrada
                sección crítica;
                lock = false;              # protocolo de salida
                sección no crítica;
```

```
        }
}
```

Solución de tipo “spin locks”: los procesos se quedan iterando (spinning) mientras esperan que se limpie lock. Cumple las 4 propiedades si el scheduling es fuertemente fair. Una política débilmente fair es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC). Spin locks tiene baja performance en multiprocesadores si varios procesos compiten por el acceso. Lock es una vble compartida y su acceso continuo es muy costoso (“memory contention”). Además, podría producirse un alto overhead por cache inválida. TS escribe siempre en lock aunque el valor no cambie → > tiempo. Mejora: Test-and-Test-and-Set:

```
while (lock) skip;              # spin mientras lock es true
while (TS(lock)){               # intenta tomar el lock
        while (lock) skip; }    # si falla, spin nuevamente
```

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

Sección crítica. Implementación de sentencias await.

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

```
SCEnter         # protocolo de entrada a la SC
S
SCExit          # protocolo de salida de la SC
```

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

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

Correcto pero ineficiente: un proceso está en spinning continuamente saliendo y entrando a la SC hasta que otro altere una vble referenciada en B.
Para reducir la contención de memoria, los procesos pueden demorarse antes de volver a intentar ganar el acceso a SC:

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

Delay podría ser un loop que itera un n° al azar de veces. Esta clase de protocolo podría usarse incluso dentro de SCEnter (en lugar de skip en el protocolo test-and-set). SI S es skip, y B cumple

ASV, < await (B); > puede implementarse **while (not B) skip;**

Soluciones 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: protocolo de SC que requiere scheduling sólo débilmente fair y no usa instrucciones especiales, pero es más complejo. Usa una vble adicional para romper empate, indicando qué proceso fue el último en comenzar a ejecutar su protocolo de entrada a la SC; **ultimo** es una variable compartida de acceso protegido. Se demora (quita prioridad) al último en comenzar su protocolo de entrada.

Tie-Breaker (grano grueso)

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

Esta solución es “casi” de grano fino: las expresiones en los await no cumplen ASV, pero no es necesario evaluar atómicamente las condiciones de demora ya que:
Si SC1 evalúa y obtiene true entonces:
- Si in2 es falso, in2 podría volverse true pero en ese caso SC2 debe haber puesto último en 2 y la condición sigue siendo tue aunque haya cambiado in2.
- Si ultimo=2, la condición se mantiene porque último cambiará sólo después que SC1 ejecute su sección crítica.
Análogamente para SC2.
Se puede implementar por busy waiting:

Tie-Breaker (grano fino)

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

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.

Algoritmo Ticket

Tie-Breaker n proceso es complejo y costoso en tiempo. En el algoritmo Ticket se reparten números y se espera turno. Los clientes toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los clientes con un número más chico sean atendidos.

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

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

Potencial problema: los valores de proximo y turno son ilimitados. En la práctica, podrían resetearse a un valor chico (por ej. 1). El predicado Ticket es un invariante global pues **numero** es leído e incrementado en una acción atómica y **proximo** es incrementado en una acción atómica; hay a lo sumo un proceso en la SC.
La ausencia de deadlock y de demor 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 **proximo** puede ser un load/store normal (a lo sumo un proceso puede estar ejecutando su protocolo de salida).
Cómo se implementa la primera acción atómica <turno[i]=número;número=número+1>?
Fetch-and-Add:

```
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: SCenter; turno[i]=numero; numero=numero+1; SCexit

Algoritmo Bakery

En Ticket, si no existe FA la solución puede no ser fair. El algoritmo Bakery es más complejo, pero es fair y no requiere instrucciones especiales. No requiere un contador global proximo que se "entrega" a cada proceso al llegar a la SC; cada proceso que trata de ingresar recorre los números de los demás y se autoasigna uno mayor. Luego espera a que su número sea el menor de los que esperan. Los procesos se chequean entre ellos y no contra un global.

```
int turno[1:n] = ([n] 0);
{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀j : 1 ≤ j
≤ n, j ≠ i: turno[j] = 0 ∨ turno[i] < turno[j] ) ) }
process SC[i = 1 to n] {
        while (true) {
                ⟨ turno[i] = max(turno[1:n] + 1; ⟩
                for [j = 1 to n st j <> i]
                ⟨ await (turno[j] == 0 or turno[i] < turno[j]); ⟩
                sección crítica
                turno[i] = 0;
                sección no crítica } }
```

Esta solución de grano grueso no es implementable directamente:
- la asignación a turno[i] exige calcular el máximo de n valores.

- el await referencia una variable compartida dos veces.

Sincronización Barrier

Problemas resueltos por algoritmos iterativos que computan sucesivas mejores aproximaciones a una rta, y terminan al encontrarla o al converger. En general manipulan un arreglo, y cada iteración realiza la misma computación sobre todos los elementos del arreglo. Múltiples procesos para computar partes disjuntas de la solución en paralelo. En la mayoría de los algoritmos iterativos paralelos cada iteración depende de los resultados de la iteración previa.
Ignorando terminación, y asumiendo n tareas paralelas en cada iteración, se tiene la forma general:

```
while (true) {
        co [i=1 to n]
                código para implementar la tarea i;
        oc }
```

Ineficiente, ya que produce n procesos en cada iteración. La idea sería crear procesos al comienzo y sincronizarlos al final de c/ iteración.

```
process Worker[i=1 to n] {
        while (true) {
                código para implementar la tarea i;
                esperar a que se completen las n tareas; }
}
```

Sincronización barrier: el punto de demora al final de c/ iteración es una **barrera** a la que deben llegar todos antes de permitirles pasar.

Sincronización Barrier: Contador Compartido

n workers necesitan encontrarse en una barrera: **cantidad** incrementado por c/ worker al llegar, cuando cantidad es n, se les permite 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: cantidad necesita ser 0 en cada iteración, puede haber contención de memoria, coherencia de cache, …

Sincronización Barrier: Flags y Coordinadores

Puede “distribuirse” cantidad usando n variables (arreglo arribo[1..n]). El await pasaría a ser ⟨ await (arribo[1] + … + arribo[n] == n); ⟩. Reintroduce memory contention y es ineficiente. Puede usarse un conjunto de valores adicionales y un proceso más; cada Worker espera por un único valor.

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);
# arribo y continuar son “flags”
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;
        }
}
```

Sincronización Barrier: Árboles

Problemas de la solución anterior: requiere un proceso (y procesador) extra. El tiempo de ejecución del coordinador es proporcional a n.
Posible solución: combinar las acciones de workers y coordinador, haciendo que cada worker sea también coordinador. Por ejemplo, workers en forma de árbol: las señales
de arribo van hacia arriba en el árbol, y las de continuar hacia abajo; combining tree barrier
(más eficiente para n grande).
En combining tree barrier los procesos juegan diferentes roles.
Una **barrera simétrica** para n procesos se construye a partir de pares de barreras simples para dos procesos:

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

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

Cómo se combinan para construir una barrera n proceso?
Worker[1:n] arreglo de procesos. Si n es potencia de 2 ⇒ butterfly barrier.

Butterfly Barrier tiene $\log_2 n$ etapas: cada Worker sincroniza con uno distinto en c/ etapa. En la etapa s, un Worker sincroniza con otro a distancia 2^{s-1}. Cuando cada Worker pasó $\log_2 n$ etapas, todos deben haber llegado a la barrera y por lo tanto todos pueden seguir.

Computaciones de Prefijo Paralelo

Algoritmos data parallel: varios procesos ejecutan el mismo código y trabajan en distintas partes de datos compartidos. Ej: computar en paralelo las sumas de los prefijos de un arreglo a[n], para obtener sum[n], donde sum[i] es la suma de los primeros i elementos de a. Secuencialmente:

```
sum[0] = a[0];
for [i=1 to n-1] sum[i] = sum[i-1] + a[i];
```

Cómo se puede paralelizar?

Idea: Sumar en paralelo pares contiguos, luego pares a distancia 2, luego a distancia 4, etc., dando un total de ($\log_2 n$) pasos.

1) Setear sum[i]=a[i]
2) En paralelo, sumar sum[i-1] a sum[i], $\forall i > 1$ (suma a distancia 1)
3) Luego, doblar la distancia, sumando sum[i-2] a sum[i], $\forall i > 2$
4) Luego de ($\log_2 n$) rondas se tienen todas las sumas parciales

valores iniciales de a[1:6]	1	2	3	4	5	6
sum después de distancia 1	1	3	5	7	9	11
sum después de distancia 2	1	3	6	10	14	18
sum después de distancia 4	1	3	6	10	15	21

Suma paralela de prefijos

```
int a[n], sum[n], viejo[n];
        process Sum[i=0 to n-1] {
        int d = 1;
        sum[i] = a[i];
        barrier(i);
        while (d<n) {
                viejo[i] = sum[i];
                barrier(i);
                if ( (i-d) >= 0 ) sum[i] = viejo[i-d] + sum[i];
                barrier(i);
                d = 2 * d;
        }
}
```

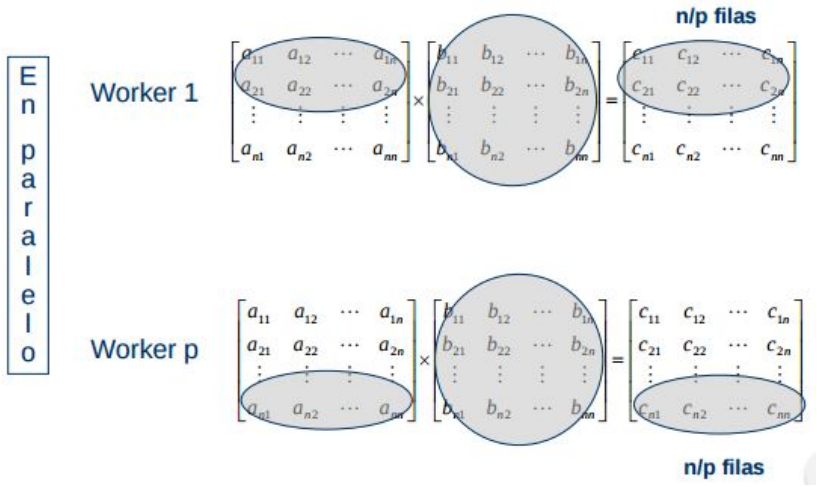
Cuál sería el efecto de usar un multiprocesador sincrónico?

Defectos de la sincronización por busy waiting

Protocolos “busy-waiting”: complejos y sin clara separación e/ vbles de sincronización y las usadas para computar resultados. Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos. Es una técnica ineficiente si se la utiliza en

multiprogramación. Un procesador ejecutando un proceso spinning puede ser usado de manera más productiva por otro proceso.
Surge la necesidad de herramientas para diseñar protocolos de sincronización.

Resolución ejercicio similar al del cuestionario



P= 8, n=128. Cuántas asignaciones, sumas y productos hace cada procesador?
Si P1=...=P7 y los tiempos de asignación son 1, de suma 2 y de producto 3; si P8 es 4 veces más lento, Cuánto tarda el proceso total? Qué puede hacerse para mejorar el speedup?

```
process worker [w = 1 to P] { # strips en paralelo (p strips de n/P filas)
        int first = (w-1) * n/P + 1        # Primera fila del strip
        int last = first + n/P - 1;        # Última fila del strip
        for [i = first to last] {
                for [j = 1 to n] {
                        c[i,j] = 0.0;
                        for [k = 1 to n]
                        c[i,j] = c[i,j] + a[i,k]*b[k,j];
                }
        }
}

process worker [w = 1 to 8] {          # 8 strips en paralelo. n/p = 16
        int first = (w-1) * 16 + 1;    # Primera fila del strip
        int last = first + 16 - 1;     # Ultima fila del strip
        # P1 = 1 a 16, P2 = 17 a 32, P3 = 33 a 48, P4 = 49 a 64
        # P5 = 65 a 80, P6 = 81 a 96, P3 = 97 a 112, P8=113 a 128
        for [i = first to last] {
                for [j = 1 to 128] {
                        c[i,j] = 0.0;      # 128 asignaciones
```

```
                        for [k = 1 to 128]
                        c[i,j] = c[i,j] + a[i,k]*b[k,j];        #128 prods, 128 sumas, 128 asign.
                }
        }
}
```

Sin considerar los incrementos de índices:
Total = 1282*16+128*16 = 264192 asig,
1282*16 = 262144 sumas
1282*16 = 262144 prod.

P1 a P8 tienen igual número de operaciones.
Total = 264192 asignaciones 262144 sumas 262144 productos
P1 =...= P7 = 264192 + 524288 + 786432 = 1574912 unid. de tiempo
Para P8 = 1574912 x 4 = 6299648 unidades de tiempo
Hay que esperar a P8 …

Si todo fuera secuencial:
1283+1282 =2113536 as. 1283 =2097152 su 1283 =2097152 prod

En P1 = ... = P7 = 2113536 + 4194304 + 6291456 = 12599296 unid. de tiempo

Speedup = 12599296 / 6299648 = 2

Si le damos a P8 solo 2 filas y a P1 a P7 18 filas, podemos corregir los tiempos:

P1=...=P7= 297216 as. 294912 sum 294912 prod
P1=...=P7= 297216 + 589824 + 884736= 1771776 unid. de tiempo

P8 con dos filas = 787456 unid. de tiempo

Speedup = 12599296 / 1771776 = 7.1

⇒ Mejor balance carga ⇒ Mejor Speedup. POR QUE NO el Speedup = 8 ?

Clase 4

Semáforo: instancia de un tipo de datos abstracto (o un objeto) con sólo 2 operaciones (métodos) atómicas: P y V. La operación V señala la ocurrencia de un evento (incrementa) y P se usa para demorar un proceso hasta que ocurra un evento (decrementa). Permiten proteger Secciones Críticas y pueden usarse para implementar sincronización por condición. Analogía con la sincronización del tránsito p/ evitar colisiones. El valor de un semáforo es no negativo.

Declaración: sem s; sem mutex = 1; sem fork[5] = ([5] 1);

Semáforo general (o counting semaphore)

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

Semáforo binario

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

Si la implementación de la demora por operaciones P se produce sobre una cola, las operaciones son fair (asumiremos que NO es así).

Problema de la Sección Crítica. Exclusión mutua

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

Semáforos. Secciones Críticas: podemos representar free con un entero, usar 1 p/ true y 0 p/ false. Se puede asociar a las operaciones soportadas por los semáforos. Es más simple que las soluciones busy waiting.

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

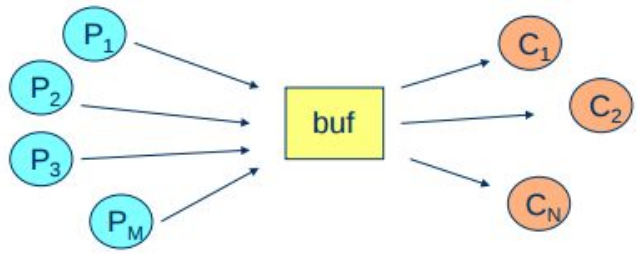
Semáforos. Barreras. Señalización de eventos
Idea: un semáforo para cada flag de sincronización. Un proceso setea el flag ejecutando V, y espera a que un flag sea seteado y luego lo limpia ejecutando P.
Barrera para dos procesos: necesitamos saber cada vez que un proceso llega o parte de la barrera, por lo que se deben 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). En una barrera p/ 2 procesos, los

eventos significativos son las llegadas de los procesos a la barrera. Se requerirán 2 semáforos de señalización.

```
sem llega1=0, llega2=0;
process Worker1 {
        .....
        V(llega1);      # señaliza el arribo
        P(llega2);      # espera al otro proceso
        ……
}
process Worker2 {
        .....
        V(llega2);      # señaliza el arribo
        P(llega1);      # espera al otro proceso
        ……
}
```

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

Productores y Consumidores. Semáforos binarios divididos (split)
Buffer unitario compartido. Múltiples productores y múltiples consumidores. Dos operaciones: depositar y retirar ⇒ deben alternarse



Dos semáforos (vacío y lleno). Inicialización: vacío en 1 y lleno en 0. Ambos semáforos binarios (por qué?).

```
typeT buf; # un buffer de algún tipo
sem vacio = 1, lleno = 0;
process Productor[i = 1 to M] {
        while(true) { …
                producir mensaje datos
                depositar: P(vacio);
                        buf = datos;
                        V(lleno)
        }
}
process Consumidor[j = 1 to N] {
```

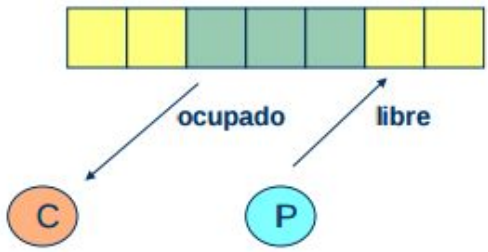
```
        while(true) {
                retirar: P(lleno);
                        resultado = buf;
                        V(vacio)
                consumir mensaje resultado
                …
        }
}
```

Los semáforos vacio y lleno, juntos, forman lo que se denomina “semáforo binario dividido” (split binary semaphore): a lo sumo uno de ellos puede ser 1 a la vez.
Split Binary Semaphore: los semáforos binarios b1, ..., bn forman un SBS en un programa si el siguiente es un invariante global:

SPLIT: $0 \leq b1, + ... + bn \leq 1$

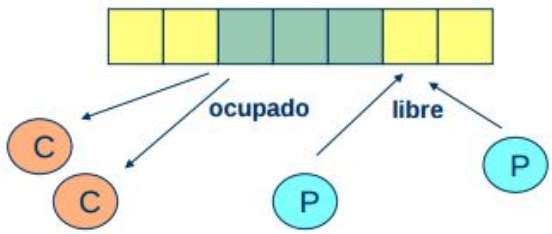
Los b_i pueden verse como un único semáforo binario b que fue dividido en n semáforos binarios.Son importantes por la forma en que pueden usarse para implementar EM (en gral la ejecución de los procesos inicia con un P sobre un semáforo y termina con un V sobre otro de ellos). Las sentencias entre el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

Buffers Limitados. Semáforos como Contadores de recursos.
Un productor y un consumidor. El buffer es una cola de mensajes depositados y aún no buscados.




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

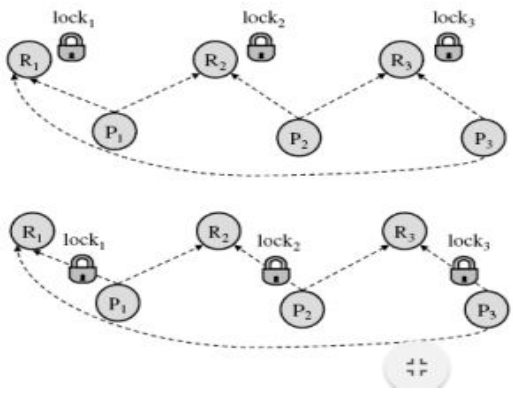
Los semáforos en este ejemplo son contadores de recursos: cada uno cuenta el número de unidades de un recurso: vacio cuenta los slots vacíos y lleno cuenta los slots llenos. Esta forma de utilización es adecuada cuando los procesos compiten por recursos de múltiple unidad. depositar y retirar se pudieron asumir atómicas pues sólo hay un productor y un consumidor.
Qué ocurre si hay más de un productor y/o consumidor? Si hay más de un productor y/o más de un consumidor, las operaciones de depositar y retirar en sí mismas son SC y deben ejecutar con EM. Cuáles serían las consecuencias de no protegerlas? Si no se protege cada slot, podría retirarse dos veces el mismo dato o perderse datos por sobreescritura.




```
typeT bu[n]; # un buffer de algún tipo
int ocupado = 0, libre = 0;
sem vacio = n, lleno = 0;
sem mutexD = 1, mutexR = 1;
process Productor [i = 1 to M] {
        while(true) { …
                producir mensaje datos
                depositar: P(vacio);
                                P(mutexD);
                                buf[libre] = datos; libre = (libre+1) mod n;
                                V(mutexD);
                                V(lleno) } }
process Consumidor[j = 1 to N] {
        while(true) {
                retirar: P(lleno);
                        P(mutexR);
                        resultado = buf[ocupado]; ocupado = (ocupado+1) mod n;
                        V(mutexR);
                        V(vacio)
                consumir mensaje resultado
… } }
```

Varios procesos compitiendo por varios recursos compartidos.

Problema de varios procesos (P) y varios recursos (R) cada uno protegido por un lock.
Un proceso debe adquirir los locks de todos los recursos que necesita. Puede caerse en deadlock cuando varios



procesos compiten por conjuntos superpuestos de recursos. Por ejemplo:
-Cada P[i] necesita R[i] y R[i+1 mod n].
-Deadlock: cada proceso adquiere lock[i] y espera por lock[i+1 mod n] que ya fue adquirido por P [i+1 mod n].

Exclusión mutua selectiva

El problema de los filósofos: ejemplo de problema de EM entre procesos que compiten por el acceso a conjuntos superpuestos de variables compartidas.

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

Cómo se especifica el adquirir y liberar los tenedores? Cada tenedor es una SC: puede ser tomado por un único filósofo a la vez. Los tenedores pueden representarse por un arreglo de semáforos:
Levantar un tenedor ⇒ P
Bajar un tenedor ⇒ V
Cada filósofo necesita el tenedor izquierdo y el derecho. Qué efecto puede darse si todos los filósofos hacen exactamente lo mismo?

```
sem tenedor[5] = {1, 1, 1, 1, 1};       # Por qué inicializados en 1??
process Filosofo [i = 0 to 3] {
        while(true) {
                P(tenedor[i]); P(tenedor[i+1]);         # toma tenedor izq. y luego der.
                come;
                V(tenedor[i]); V(tenedor[i+1]);
                piensa;
        }
}
process Filosofo [4] {
        while(true) {
                P(tenedor[0]); P(tenedor[4]);           # toma tenedor der. y luego izq.
                come;
                V(tenedor[0]); V(tenedor[4]);
                piensa;
        }
```

}

Lectores y Escritores: 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: (1) como problema de exclusión mutua y (2) como problema de sincronización por condición.

Lectores y Escritores como problema de Exclusión mutua: los escritores necesitan acceso mutuamente exclusivo. Los lectores (como grupo) necesitan acceso exclusivo con respecto a cualquier escritor.

Solución demasiado **restrictiva**:

```
sem rw = 1;
process Lector [i = 1 to M] {
        while(true) { …
                P(rw);          # toma bloqueo de acceso exclusivo
                lee la BD;
                V(rw);          # libera el bloqueo
        } }
process Escritor [j = 1 to N] {
        while(true) { …
                P(rw);          # toma bloqueo de acceso exclusivo
                escribe la BD;
                V(rw);          # libera el bloqueo
        } }
```

Los lectores (como grupo) necesitan bloquear a los escritores, pero sólo el primero necesita tomar el lock ejecutando P(rw). Análogamente, sólo el último lector debe hacer V(rw).

```
int nr = 0; # número de lectores activos
sem rw = 1;
process Lector [i = 1 to M] {
        while(true) { …
        ⟨ nr = nr + 1;
          if (nr == 1) P(rw); ⟩          # si es el primero, toma el bloqueo
        lee la BD;
        ⟨ nr = nr - 1;
          if (nr == 0) V(rw); ⟩          # si es el último, libera el bloqueo } }

process Escritor [j = 1 to N] {
        while(true) { …
                P(rw);          # toma bloqueo de acceso exclusivo
                escribe la BD;
                V(rw);          # libera el bloqueo } }
```

Solución final:

```
int nr = 0;             # número de lectores activos
sem rw = 1;             # bloquea acceso a la BD
sem mutexR= 1;          # bloquea acceso de los lectores a nr
process Lector [i = 1 to M] {
        while(true) { …
                P(mutexR);
                        nr = nr + 1;
                        if (nr == 1) P(rw);       # si es el primero, toma el bloqueo
                V(mutexR);
                lee la BD;
                P(mutexR);
                        nr = nr - 1;
                        if (nr == 0) V(rw);       # si es el último, libera el bloqueo
                V(mutexR); } }
process Escritor [j = 1 to N] {
        while(true) { …
                P(rw);          # toma bloqueo de acceso exclusivo
                escribe la BD;
                V(rw);          # libera el bloqueo } }
```

Lectores y Escritores usando Sincronización por Condición: la solución anterior da preferencia a los lectores, por lo que no es fair. Además, no es sencillo modificarla. Otro enfoque ⇒ introduce la técnica **passing the baton**: emplea SBS para brindar exclusión y despertar procesos demorados. Puede usarse para implementar *await* arbitrarios, controlando de forma precisa el orden en que los procesos son despertados. En este caso, pueden contarse los procesos de cada clase intentando acceder a la BD, y luego restringir el valor de los contadores.
nr y nw enteros no negativos que registran el número de lectores y escritores accediendo a la BD.
Estados malos a evitar: nr y nw positivos, o nw mayor que 1:
BAD: (nr > 0 AND nw > 0) OR nw > 1
Estados buenos caracterizados por NOT BAD:
RW: (nr == 0 OR nw == 0) AND nw <= 1

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

La técnica Passing the Baton

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 E/ 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: técnica general para implementar sentencias await.

Passing the baton: cuando un proceso está dentro de una SC mantiene el baton (testimonio, token) que significa permiso para ejecutar. Cuando el proceso llega a un SIGNAL, pasa el baton (control) a otro proceso. Si ningún proceso está esperando una condición que sea true, el baton se pasa al próximo proceso que trata de entrar a su SC por primera vez (es decir, uno que ejecuta P(e)).

Lectores y escritores:

- e semáforo p/ controlar acceso a las vbles compartidas, r semáforo asociado a la guarda en procesos lectores, y w asociado a la guarda en escritores.
- dr y dw contadores de lectores y escritores esperando.

```
int nr = 0, nw = 0;
sem e = 1, r = 0, w = 0: # siempre 0 <= (e+r+w) <= 1
int dr = 0, dw = 0;
process Lector [i = 1 to M] {
        while(true) {
                # ⟨ await (nw == 0) nr = nr + 1; ⟩
                P(e);
                if (nw > 0) {dr = dr+1; V(e); P(r); }
                nr = nr + 1;
                if (dr > 0) {dr = dr – 1; V(r); }              #signal
                else V(e);
                lee la BD;
                # ⟨ nr = nr – 1; ⟩
                P(e);
                nr = nr – 1;
                if (nr == 0 and dw > 0) {dw = dw – 1; V(w); }          #signal
                else V(e);
        }
}

process Escritor [j = 1 to N] {
        while(true) {
                # ⟨ await (nr == 0 and nw == 0) nw = nw + 1; ⟩
                P(e);
                if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
                nw = nw + 1;
                V(e);                       #signal
                escribe la BD;
```

```
                # ⟨ nw = nw – 1; ⟩
                P(e);
                nw = nw – 1;
                if (dr > 0) {dr = dr - 1; V(r); }
                elseif (dw > 0) {dw = dw - 1; V(w); }          #signal
                else V(e);
        }
}
```

Da preferencia a los lectores, cómo puede modificarse?

Alocación de Recursos y Scheduling

Cómo 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.
Hasta ahora aplicamos la política más simple: si algún proceso está esperando y el recurso está disponible, se lo asigna. La política de alocación más compleja fue en R/W, aunque daba preferencia a clases de procesos, no a procesos individuales. Cómo se pueden implementar políticas de alocación de recursos generales y cómo controlar explícitamente cuál proceso toma un recurso si hay más de uno esperando?

Definición del problema y patrón de solución general: 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: 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. Cuando el recurso es liberado(release ( )), 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.
Ejemplos:

- alocación de procesador (tiempo es el tiempo de ejecución).
- spooling de una impresora (tiempo tiempo de impresión).

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 (esto es, si libre es falso).

```
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.

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. b[1:n] es un arreglo de semáforos, donde cada entry es inicialmente 0 ⇒ el proceso id se demora sobre el semáforo b[id].

```
bool libre = true;
sem e = 1, b[n] = ([n] 0);                  # para entry y demora
typedef Pares = set of (int, int);
Pares = ∅ ;
# SJN: Pares es un conjunto ordenado ∧ libre ⇒ ( pares== ∅ )

request(tiempo,id):     P(e);
                        if (! free) {
                                insertar (tiempo,id) en Pares;
                                V(e);           # libera el bloqueo
                                P(b[id]);       # espera en SU semáforo a ser despertado }
                        libre = false;          # Se está tomando el recurso
                        V(e);                   # libera el bloqueo p/ encolar otros procesos
release( ):     P(e);
                libre = true;
                if (Pares ≠ ∅ ) {
```

```
            remover el primer par (tiempo,id) de Pares;
            V(b[id]);                    # pasa el baton al proceso id }
        else V(e);
```

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

Clase 5 y 6

Características de los Semáforos:
-Variables compartidas globales a los procesos.
-Sentencias de control de acceso a la SC dispersas en el código.
-Al agregar procesos, se debe verificar acceso correcto a las VC.
-Aunque EM y SxC son conceptos ≠, se programan de forma similar.

Monitores: módulos de programación con más estructura, y que pueden ser implementados tan eficientemente como los semáforos. Son un mecanismo de abstracción de datos: encapsulan las representaciones de objetos (recursos) y brindan un conjunto de operaciones que son los únicos medios para manipular la representación. Contiene variables que almacenan el estado del recurso y procedimientos que implementan las operaciones sobre él.
La EM es implícita en monitores, asegurando que los procedures en el mismo monitor no ejecutan concurrentemente. La SxC es explícita con variables condición. En un programa concurrente, los procesos son activos y los 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.

Un monitor agrupa la representación y la implementación de un recurso compartido (clase). Lo que distingue a un monitor de un TAD es que puede ser compartido por procesos concurrentes. Tiene interfaz y cuerpo; la interfaz especifica operaciones (métodos) que brinda el recurso, el cuerpo tiene vbles que representan el estado del recurso y procedures
que implementan las operaciones de la interfaz. Sólo los nombres de los procedures son visibles desde afuera. Sintácticamente, los llamados al monitor tienen la forma:

call NombreMonitor.op$_i$ (argumentos)

Los procedures pueden acceder sólo variables permanentes, sus vbles locales, y parámetros que le sean pasados en la invocación. El programador de un monitor no puede conocer a priori el orden de llamado, pero puede definirse un invariante del monitor que especifica los estados “razonables” de las vbles permanentes cuando ningún proceso las está accediendo.

Notación:

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

{ cuerpo de op_1 }

.....

procedure op_n (par. $formales_n$)

{ cuerpo de op_n }

}

Sincronización

La exclusión mutua en monitores se provee implícitamente. La SxC es programada explícitamente con variables condición (*cond cv;*). El valor asociado a cv es una cola de procesos demorados, no visible directamente al programador.
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 y lo saca de ella (cola vacía = skip). Ese proceso puede ejecutar cuando readquiera el acceso exclusivo al monitor (depende de la disciplina de señalización).
signal_all(cv) → despierta todos los procesos demorados en cv. El tiempo en que cada uno reinicie efectivamente la ejecución dependerá de las condiciones de exclusión mutua.

Signal and continue: el proceso que ejecuta SIGNAL retiene el control exclusivo del monitor y puede seguir ejecutando, mientras el proceso despertado pasa a competir por acceder nuevamente al monitor y continuar su ejecución en la instrucción siguiente al wait (Unix, Java, Pthreads).
Signal and wait: el proceso que hace SIGNAL pasa a competir por acceder nuevamente al monitor, mientras que el despertado pasa a ejecutar dentro del monitor en la instrucción siguiente al wait.
WAIT y SIGNAL son similares a P y V, pero hay diferencias:

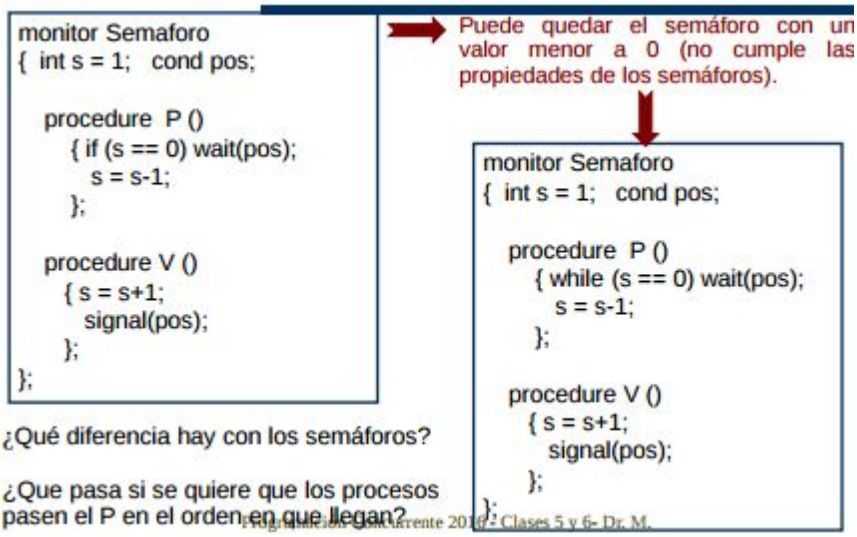
WAIT	P
El proceso siempre se duerme	El proceso sólo se duerme si el semáforo es 0.
SIGNAL	**V**
Si hay procesos dormidos despierta al primero de ellos. En caso contrario no tiene efecto posterior.	Incrementa el semáforo para que un proceso dormido o que hará un P continúe. No sigue ningún orden al despertarlos.

Operaciones adicionales en Monitores

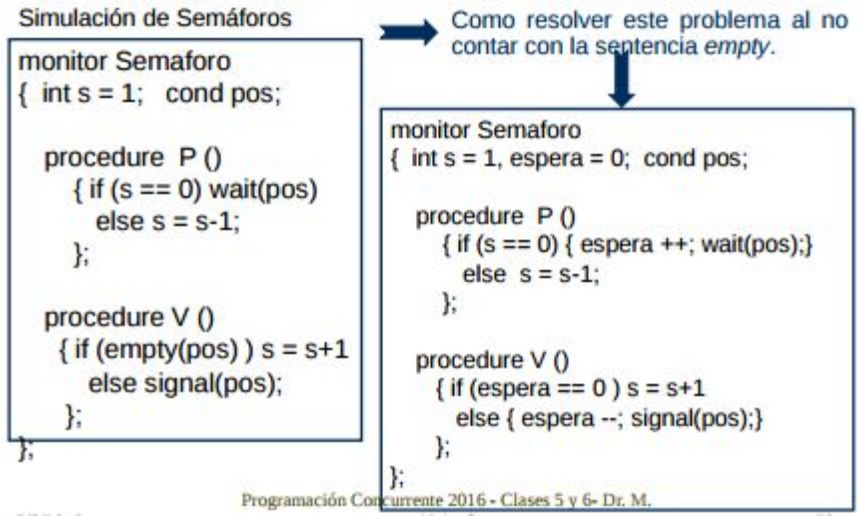
empty(cv)→ retorna true si la cola controlada por cv está vacía.
wait(cv,rank) → wait con prioridad: permite tener más control sobre el orden en que los procesos son encolados y despertados. Los procesos demorados en cv son despertados en orden ascendente de rank.
minrank(cv) → función que retorna el mínimo ranking de demora. Con semántica S&C, signal_all es lo mismo que while (not empty(cv)) signal(cv);

Estas operaciones no son usadas en la práctica de la materia.

Ejemplo – Simulación de semáforos: condición básica



Técnicas de sincronización – Simulación de semáforos: passing the condition



Monitores. Alocación SJN

```
monitor Shortest_Job_Next {
bool libre = true;
cond turno; # Signal cuando recurso está disponible
procedure request(int tiempo) {
 if (libre)
   libre = false;
 else # lo duermo ordenado x tiempo
   wait(turno, tiempo);
 }

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

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

Cómo se resuelve sin Wait con prioridad? Manejo del orden explícitamente usando una cola ordenada y variables condición privadas:

```
monitor Shortest_Job_Next
{ bool libre = true;
  cond turno[N];
  cola  espera;

  procedure request (int id, int tiempo)
     { if (libre) libre = false
       else { insertar_ordenado(espera, id, tiempo);
              wait (turno[id]);
            };
     };

  procedure release ()
    { if (empty(espera)) libre = true
      else  { sacar(espera, id);
              signal(turno[id]);
            };
    };
}
```

Monitores. Buffer limitado

```
monitor Buffer_Limitado {
  typeT buf[n];              # array de algún tipo T
  int ocupado = 0,           # índice del primer slot lleno
      libre = 0;             # índice del primer slot vacío
      cantidad = 0;          # cantidad de slots llenos
                             # rear == (front + count) mod n
  cond not_lleno,            # signal cuando count < n
       not_vacio;            # signal cuando count > 0
  procedure depositar(typeT datos) {
      while (cantidad == n) wait (not_lleno);
      buf[libre] = datos; libre = (libre+1) mod n; cantidad++;
      signal(not_vacio);   }
  procedure retirar(typeT &resultado) {
      while (cantidad == 0) wait(not_vacio);
      resultado=buf[ocupado]; ocupado=(ocupado+1) mod n; cantidad--;
      signal(not_lleno);       }
}
```

Lectores y Escritores con monitores. Broadcast signal: el monitor arbitra el acceso a la BD. Los procesos dicen cuándo quieren acceder y cuándo terminaron; requieren un monitor con 4 procedures: pedido_leer, libera_leer, pedido_escribir y libera_escribir. nr es el número de lectores y nw el número de escritores (variables permanentes). Invariante: RW: (nr = 0 ∨ nw = 0) ∧ nw ≤ 1.

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

Clase 7

Programación Distribuida. Conceptos básicos.

Programa distribuido: programa concurrente comunicado por mensajes. Supone la ejecución sobre una arq. de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida). Los procesos comparten canales (físicos o lógicos).
Los canales son lo único que comparten los procesos. Las variables son locales a un proceso. La EM no requiere mecanismo especial, los procesos interactúan comunicándose. Los canales son accedidos por primitivas de envío y recepción.

Clasificaciones de los canales

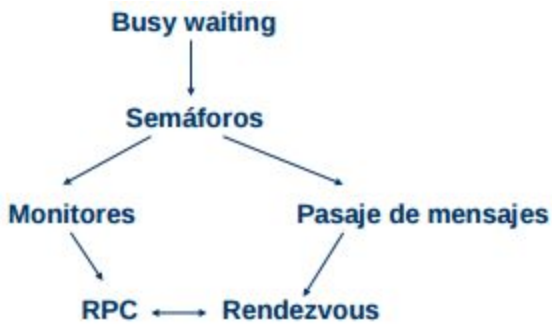
De acuerdo a quién recibe y envía:
- Mailbox: cualquiera puede enviar y recibir.
- Input port: sólo uno puede recibir por ese canal.
- Link: sólo un proceso puede enviar y sólo un proceso puede recibir.

De acuerdo a si se puede enviar y recibir por la misma comunicación:
- Uni o bidireccionales.

De acuerdo al momento en que los procesos pueden continuar:
- Sincrónicos o asincrónicos.

Programación Distribuida. Combinaciones: mecanismos “equivalentes” funcionalmente: - PMA - PMS - RPC - Rendezvous. La sincronización de la comunicación interproceso depende del patrón de interacción:
- Productores y consumidores (Filtros, Pipes). Ejemplo: sorting network.
- Clientes y servidores. Ejemplos: alocación de recursos, file servers, scheduling.
- Peers. Ejemplos: probe/echo, heartbeat, semáforos distribuidos, servers replicados.
Cada mecanismo es más adecuado para determinados patrones.

Relación entre mecanismos de sincronización
Semáforos ⇒ mejora respecto de busy waiting;
Monitores ⇒ combinan EM implícita y señalización explícita;
PM ⇒ extiende semáforos con datos;
RPC y rendezvous ⇒ combinan la interfase procedural de monitores con PM implícito.



<u>Pasaje de Mensajes Asincrónicos</u>

Los **canales** son colas de mensajes enviados y aún no recibidos. Un proceso agrega un mensaje al

final de la cola (“ilimitada”) de un canal ejecutando un send, que no bloquea al emisor. Un proceso recibe un msg desde un canal con receive, que demora al receptor hasta que en el canal haya al menos un msg; luego toma el primero y lo almacena en variables locales. Las variables del receive deben tener los mismos tipos que la declaración del canal. Receive es una primitiva bloqueante, ya que produce un delay. El proceso NO hace nada hasta recibir un msg en la cola correspondiente al canal ch; NO es necesario hacer polling.

El acceso a los contenidos de c/ canal es 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”.
La instrucción 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; la evaluación de empty podría ser true, y sin embargo existir un mensaje al momento de que el proceso reanuda la ejecución, o podría ser false, y no haber más mensajes cuando sigue ejecutando (si no es el único en recibir por ese canal).

Ejemplo de proceso filtro:


```
chan entrada(char), salida(char [CantMax]);   # Canales globales
Process Carac_a_Linea { char linea [CantMax], int i := 0;
        WHILE true {
                receive entrada (linea[i]);
                WHILE linea[i] ≠ CR and i < CantMax {
                        i := i + 1;
                        receive entrada (linea[i]);  }
                linea [i] := EOL;
                send salida(linea);
                i := 0 }
  }
```

En el ejemplo anterior los canales entrada y salida son declarados globales a los procesos, ya que pueden ser compartidos. 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.

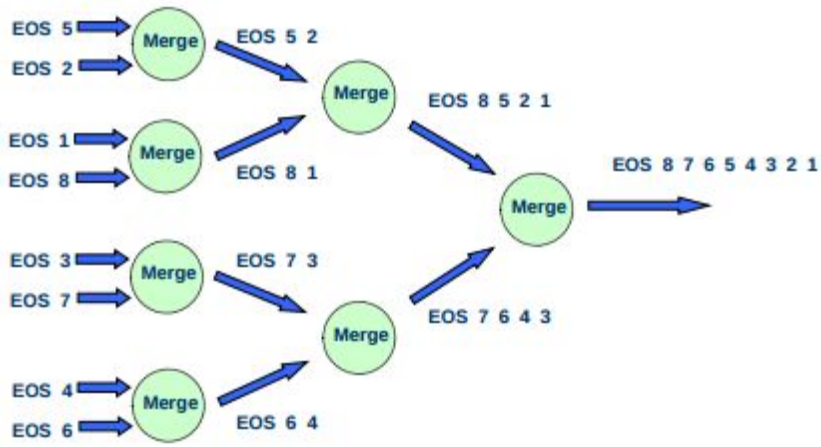
Filtros: Red de Ordenación

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

Problema: ordenar una lista de N números de modo ascendente. Podemos pensar en un filtro Sort con un canal de E/ (N números desordenados) y un canal de S/ (N números ordenados).

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

Problema: cómo determina Sort que recibió todos los números? (1) conoce N (2) envía N como el primer elemento a recibir por el canal entrada o (3) 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 N1 elementos cada una en una lista ordenada de 2*N1 elementos. Con PMA, pensamos en 2 canales de E/ por cada canal de S/. Un carácter especial EOS cerrará cada lista parcial ordenada.
La red es construida con filtros Merge. C/ Merge recibe valores de dos streams de E/ 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.



Se envían n*$\log_2$n mensajes.
Hay n-1 procesos; el ancho de la red es $\log_2$n. Canales de E/ y S/ compartidos. Puede programarse usando:
- static naming (arreglo global de canales, y c/ 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 c/ 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 S/ de uno cumpla las suposiciones de E/ del otro ⇒ pueden reemplazarse si se mantienen los comportamientos de E/ y S/.

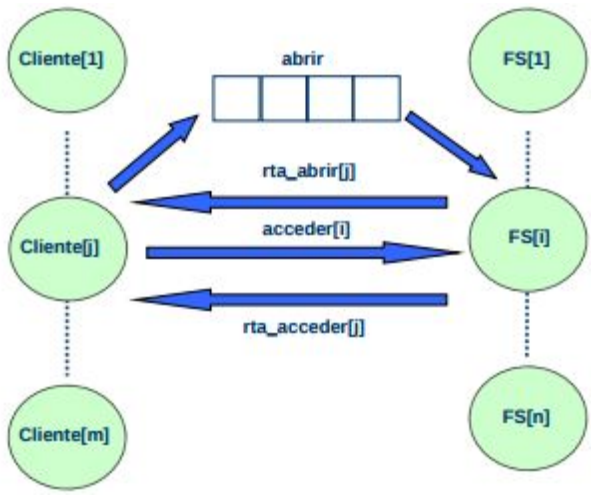
Dualidad entre Monitores y Pasaje de Mensajes

Programas con Monitores	**Programas basados en PM**
Variables permanentes	Variables locales
Identificadores de procedures	Canal request y tipos de operación
Llamado a procedure	send request( ); receive respuesta
Entry del monitor	receive request( )
Retorno del procedure	send respuesta( )
Sentencia wait	Salvar pedido pendiente
Sentencia signal	Recuperar/ procesar pedido pendiente
Cuerpos de los procedure	Sentencias del “case” de acuerdo a la clase de operación

La eficiencia de monitores o PM depende de la arq. 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.

Continuidad Conversacional

Ejemplo: procesos “cliente” que acceden a archivos externos almacenados en disco. Deben hacer OPEN; si el archivo se puede abrir hacen una serie de pedidos de READ o WRITE y luego cierran el archivo (CLOSE). Si hay N archivos, consideramos 1 File Server por archivo. Los procesos servidores son idénticos, y cualquiera de ellos que esté libre puede atender un requerimiento de OPEN.
Todos los clientes pueden pedir OPEN por un canal global (qué argumentos son necesarios?) y recibirán respuesta de un servidor dado por un canal propio (por qué? dos proc no podrían hacer recieve al mismo tiempo porque el recieve es atómico).

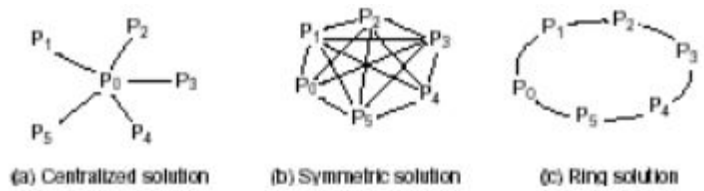


Este ejemplo de interacción entre clientes y servidores se denomina **continuidad conversacional** (del ABRIR al CERRAR). *abrir* es un canal compartido por el que cualquier FS puede recibir. Si c/ canal puede tener 1 solo receptor, necesito un alocador de archivos separado, que recibiría pedidos de ABRIR y alocaría un FS libre a 1 cliente; los FS necesitarían decirle al alocador cuándo

están libres. Si el lenguaje soporta creación dinámica de procesos y canales, el número n de FS puede adaptarse a los requerimientos del sistema. Otro esquema sería un file server por disco (interfase más compleja). Otra solución: Sun Netwok File System: ABRIR ⇒ adquirir un descriptor completo del file. Las operaciones sucesivas son RPC transmitiendo el descriptor (ventajas y desventajas).

PMA. Pares (peers) interactuantes: Intercambio de valores

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



La 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.
Se envían 2(N-1) mensajes. Si p[0] dispone de una primitiva broadcast se reduce a N mensajes.

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

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

Comentarios sobre las soluciones

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

Centralizada y **anillo** usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance.
En **centralizada**, los msgs al coordinador se envían casi al mismo tiempo; sólo el 1er receive del coordinador demora mucho.
En **anillo**, todos los procesos son productores y consumidores. El último tiene que esperar a que todos los otros (uno por vez) reciban un msg, hacer poco cómputo, y enviar su resultado. Los msg circulan 2 veces completas por el anillo ⇒ Solución inherentemente lineal y lenta para este problema, pero puede funcionar si cada proceso tiene mucho cómputo.

Algoritmo Heartbeat: se expande, enviando información; luego se contrae, incorporando nueva información. Útil para el cálculo de la topología de una red.

Clase 8

Pasaje de Mensajes Sincrónicos: La principal diferencia con PMA es que la primitiva de transmisión (llamemosla sync_send) es bloqueante; el trasmisor queda esperando que el mensaje sea recibido. La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje, por lo que utiliza MENOS memoria. Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (siempre un proceso se bloquea). Los canales son punto a punto (1 emisor – 1 receptor). Si bien send y sync_send son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de deadlock son mayores en comunicación sincrónica.
Ejemplo: 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
    }
}
```

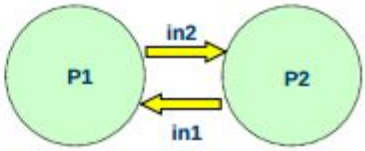
Comentarios: si los cálculos del productor se realizan mucho más rápido que los del consumidor en las primeras n1 operaciones, y luego se realizan mucho más lento durante otras n1 interacciones:
- Con PMS los pares send/receive se completarán asumiendo la demora del proceso que más tiempo consuma. Si la relación de tiempo fuera 10-1 significaría multiplicar por 10 los tiempos totales.
- Con PMA, al principio el productor es más rápido y sus mensajes se encolan. Luego el consumidor es más rápido y “descuenta” tiempo consumiendo la cola de mensajes.

Conclusión: mayor concurrencia en AMP. Para lograr el mismo efecto en PMS se debe interponer un proceso “buffer”.

La concurrencia también se reduce en algunas interacciones C/S:
- 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 y receive hagan matching.

Dos procesos que intercambian valores:

```
chan in1(INT), in2(INT);
Process P1  {
    INT value1 = 1, value2;
    sync_send in2(value1);
    receive in1 (value2);
}
Process P2  {
    INT value1, value2=2;
    sync_send in1(value2);
    receive in2 (value1);
}
```



Con mensajes sincrónicos esta solución entra en deadlock (por qué?). Con AMP esta resolución es válida. (Además, al ser simétrica es escalable fácilmente).

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

Formas generales de las sentencias de comunicación:

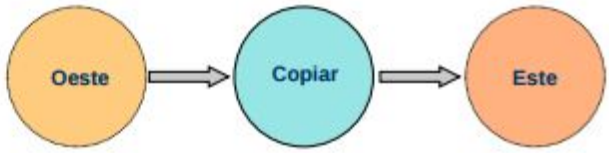
Destino ! port(e1, ..., en);
Fuente ? port(x1, ..., xn);

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

A ! canaluno(dato); B ? canaluno(resultado);

Ejemplo: proceso filtro que copia caracteres recibidos del proceso Oeste al proceso Este:

```
Process Copiar {
    CHAR c;
    do true →
        Oeste ? c;
        Este ! c ;
    od
}
```



Las operaciones de comunicación (? y !) pueden ser guardadas (equivalente a un AWAIT hasta que una condición sea verdadera). El do e if de CSP usan los comandos guardados de Dijkstra (B → S).

CSP. Comunicación Guardada

Limitaciones de ? y ! ya que son bloqueantes. 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 sentencias de comunicación guardada soportan comunicación no determinística: B; C → S;

B puede omitirse y se asume true. B y C forman la **guarda**. Una guarda puede tener uno de tres estados:

La guarda tiene éxito si B es true y ejecutar C no causa demora.

La guarda falla si B es falsa.

La guarda se bloquea si B es true pero C no puede ejecutarse inmediatamente.

Las sentencias de comunicación guardadas aparecen en if y do.

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

Ejecución:

Primero, se evalúan las expresiones booleanas

- Si ambas guardas fallan, el if termina sin efecto.
- Si al menos una guarda tiene éxito, se elige una (no deterministicamente).
- Si ambas guardas se bloquean, se espera hasta que una tenga éxito.

Segundo, luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación en la guarda elegida.

Tercero, se ejecuta la sentencia S_i.

La ejecución del **do** es similar (se repite hasta que todas las guardas fallen).

Intercambio de valores entre dos procesos:

```
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.

Generación de números primos: La Criba de Eratóstenes.
Problema: generar todos los primos entre 2 y n. Comenzando con el primer número (2), recorremos la lista y borramos los múltiplos de ese número. 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? un proceso por cada número (mala solución...) o un 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. El número total de procesos 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 P1 y P2, cada uno inicialmente con n/2 valores (arreglos a1 y a2 respectivamente). Los n/2 valores de cada proceso se encuentran ordenados inicialmente.

Idea: realizar una serie de intercambios. En cada uno P1 y P2 intercambian a1[mayor] y a2[menor], hasta que a1[mayor] < a2[menor].

Solución con k procesos P[1:k], inicialmente con n/k valores c/u. Cada uno primero ordena sus n/k 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 P1, y cada proceso con número par el de P2.
- En las rondas pares, cada proceso numerado par juega el rol de P1, y cada proceso impar el rol de P2.
Cada intercambio progresa 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. 2k mensajes de overhead en cada ronda.
- Que cada proceso ejecute suficientes rondas para garantizar que la lista estará ordenada (en general, al menos k rondas). En el k-proceso, cada uno intercambia hasta n/k+1 msg por ronda. El algoritmo requiere hasta $k^2*(n/k + 1)$ intercambio de mensajes.

El lenguaje OCCAM: Hoare introdujo CSP como lenguaje formal que sigue el modelo de PMS, pero nunca fue implementado en forma completa. OCCAM es un lenguaje real, que implementa lo esencial de CSP sobre la arquitectura "modelo" de los transputers. Transputers + OCCAM = "sistema multiprocesador p/ procesamiento concurrente", donde tanto la arquitectura como el lenguaje son simples y adecuadas a la programación concurrente PMS.
OCCAM es un lenguaje simple con una sintaxis rígida. Los procesos y los caminos de comunicación e/ ellos son estáticos (cantidades fijas definidas en compilación). Modelo de comunicación sincrónico por canales half duplex. Las sentencias básicas (asignación y comunicación) son vistas como procesos primitivos. No soporta recursión, ni creación o nombrado dinámico; algunos algoritmos son difíciles de programar, aunque el compilador puede determinar cuántos procesos tiene un programa y cómo se comunican. Permite mapear procesos a procesadores del transputer.

Clase 9

Programación Paralela con el concepto de Bag of Tasks

Idea: tener una "bolsa" de tareas que pueden ser compartidas por procesos "worker". C/ worker ejecuta un código básico:

```
while (true) {
        obtener una tarea de la bolsa
        if (no hay más tareas)
                BREAK; # exit del WHILE
        ejecutar tarea (incluyendo creación de tareas);
}
```

Puede usarse para resolver problemas con un n° fijo de tareas y para soluciones recursivas con nuevas tareas creadas dinámicamente. El paradigma de "bag of tasks" es sencillo, escalable (aunque no necesariamente en performance) y favorece el balance de carga entre los procesos.

Multiplicación de matrices con Bag of Tasks

Multiplicación de 2 matrices a y b de nxn ⇒ n^2 productos internos e/ filas de a y columnas de b. Cada producto interno es independiente y se puede realizar en paralelo. Suponiendo una máquina con PR procesadores (PR < n)⇒ PR procesos worker. Para balancear la carga, c/u debería computar casi el mismo número de productos internos (buscando trabajo cuando no tiene). Si PR << n, un buen tamaño de tarea sería una o unas pocas filas de la matriz resultado c. Por simplicidad, suponemos 1 fila. Inicialmente, la bolsa contiene n tareas, una por fila. Pueden estar ordenadas de cualquier manera ⇒ se puede representar la bolsa, simplemente contando filas: INT proxfila = 0; Un worker saca una tarea de la bolsa con: donde fila es una variable local (Fetch and

Add).

LINDA: aproximación distintiva al procesamiento concurrente que combina aspectos de MC y PMA. NO es un lenguaje de programación, sino un conjunto de 6 primitivas que operan sobre una MC donde hay "tuplas nombradas" (tagged tuples) que pueden ser pasivas (datos) o activas (tareas). Puede agregarse como biblioteca a un lenguaje secuencial. El núcleo de LINDA es el espacio de tuplas compartido (TS) que puede verse como un único canal de comunicaciones compartido, pero en el que no existe orden:
- Depositar una tupla (OUT) funciona como un SEND.
- Extraer una tupla (IN) funciona como un RECEIVE.
- RD permite "leer"como un RECEIVE pero sin extraer la tupla de TS.
- EVAL permite creación de procesos (tuplas activas) dentro de TS.
- INP y RDP permiten hacer IN y RD no bloqueantes.

Si bien hablamos de MC, TS puede estar físicamente distribuida en una arq. multiprocesador (más complejo) ⇒ puede usarse para almacenar estr. de datos distribuidas, y ≠ procesos pueden acceder concurrentemente diferentes elementos de las mismas.

Conceptos de RPC y Rendezvous

El PM se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la **comunicación unidireccional**. Para resolver 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**, lo que los hace 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: difieren en la manera de servir la invocación de operaciones:
- Un enfoque es declarar un procedure p/ c/ operación y crear un nuevo proceso (al menos conceptualmente) p/ manejar c/ llamado (RPC porque el llamador y el cuerpo del procedure pueden estar en distintas máquinas. P/ 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 E/ (o accept) que espera una invocación, la procesa y devuelve los resultados (Ej: Ada).

Rendezvous extendido, para diferenciarlo del simple de PMS. Modelo de CSP extendido p/ tratar procesos dinámicos y bidireccionales.

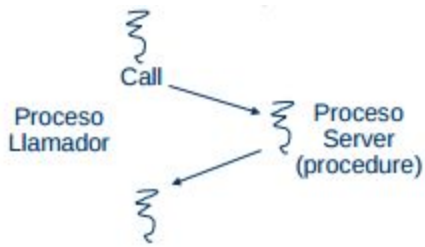
Remote Procedure Call: los programas se descomponen en módulos (con procesos y procedures), que pueden residir en espacios de direcciones distintos. Los procesos de un módulo pueden compartir variables y llamar a procedures de ese módulo. Un proceso en un módulo puede

comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste. Los módulos tienen especificación e implementación de procedures:

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

Los procesos locales son llamados background para distinguirlos de las operaciones exportadas.
Header de un procedure visible:
op opname (formales) [**returns** result].
El cuerpo de un procedure visible es contenido 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 intermódulo es distinta que p/ 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 e/ 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 gral, un llamado será remoto ⇒ se debe crear un proceso server o alocarlo de un pool preexistente.



Sincronización en Módulos

Por sí mismo, RPC es **solo un mecanismo de comunicación**. Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador (como si éste estuviera ejecutando el llamado ⇒ la sincronización entre ambos es implícita). Necesitamos que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo). Esto comprende EM y SxC.
Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan con exclusión mutua (un solo proceso por vez) o concurrentemente.

Si ejecutan con **EM** las VC son protegidas automáticamente contra acceso concurrente, pero es necesario programar SxC.
Si pueden ejecutar **concurrentemente** necesitamos mecanismos para programar EM y SxC **(c/ módulo es un programa concurrente)** ⇒ podemos usar cualquier método ya descripto (semáforos, monitores, o incluso rendezvous).
Es más general asumir que los procesos pueden ejecutar concurrentemente (más eficiente en un multiprocesador de MC).

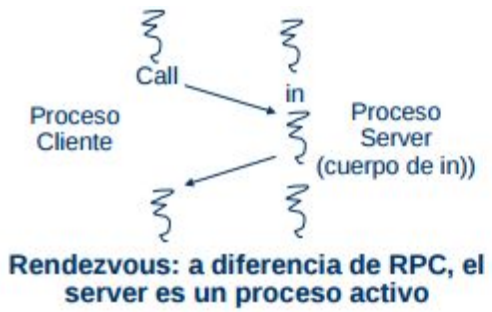
Rendezvous

RPC por si mismo sólo brinda un mecanismo de comunicación intermódulo. Dentro de un módulo es necesario programar la sincronización. Además, a veces son necesarios procesos extra sólo para manipular los datos comunicados por medio de RPC (ej: Merge).
Rendezvous combina comunicación y sincronización:
- Como con RPC, un proceso cliente **invoca** una operación por medio de un **call**, pero esta operación es **servida por un proceso existente** en lugar de por uno nuevo.
- Un proceso servidor usa una **sentencia de entrada** para esperar por un call y actuar.
- Las operaciones se atienden una por vez más que concurrentemente.

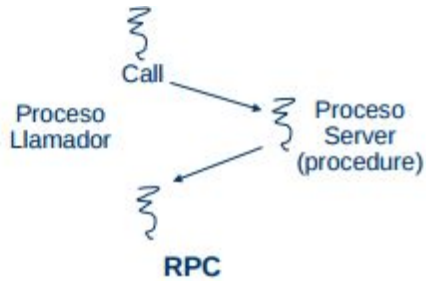
La especificación de un módulo contiene declaraciones de los headers de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones. Si un módulo exporta **opname**, el proceso server en el módulo realiza *rendezvous* con un llamador de **opname** ejecutando una sentencia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

in opname(identif. formales) → **S; ni**

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



Rendezvous: a diferencia de RPC, el server es un proceso activo



RPC

Combinando comunicación guardada con rendezvous:

in op_1 ($formales_1$) and B_1 by e_1 → S_1;
□ ...
□ op_n ($formales_n$) and B_n by e_n → S_n;
ni

Las B_i son **expresiones de sincronización** opcionales. Los e_i **son expresiones de scheduling** opcionales. En ambos casos, pueden referenciar a los parámetros formales.

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

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 en ADA. Call: Entry call.
El rendezvous es el principal mecanismo de sincronización en Ada y también es el mecanismo de comunicación primario. Las declaraciones de entry son similares a las de op:

entry identificador (formales)

Los parámetros del entry pueden ser IN, OUT o IN OUT. También soporta arreglos de entries, llamados familias de entry.
Entry call: Si el task T declara el entry E, otras tasks en el alcance de la especificación de T pueden invocar a E con: **call T.E (parámetros reales)**. La ejecución del call demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).
Se usa **entry call condicional** si una tarea quiere hacer polling de otra:

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

Elige el entry call si puede ejecutarse de inmediato; sino, elige else.
Se usa **entry call temporal** (timed entry call) si una tarea llamadora quiere esperar a otra a lo sumo un intervalo de tiempo:
select entry call;

sentencias adicionales;
or delay tiempo;
sentencias
end select;

Sentencia de Entrada: Accept
La task que declara un entry sirve llamados al entry con accept:
accept nombre (parámetros formales) **do** sentencias **end;**
Demora la tarea hasta que haya una invocación, copia los parámetros reales en los formales, y ejecuta las sentencias. Cuando termina, los parámetros formales de S/ son copiados a los parámetros reales. Luego, el llamador y el proceso ejecutante continúan.

accept E1(parámetros formales) do cuerpo de E1 end E1;
accept E2(parámetros formales) do cuerpo de E2 end E2;
accept E3(parámetros formales) do cuerpo de E3 end E3;

Especifica que se espera un pedido por el entry E1, luego de atendido se espera un pedido por E2 y luego un pedido por E3.

La **sentencia wait selectiva** soporta comunicación guardada:

select when B_1 ⇒ **accept** E_1; $sentencias_1$
or ...
or when B_n ⇒ **accept** E_n; $sentencias_n$
end select

Cada línea (salvo la última) se llama alternativa. B_i son expr. booleanas, y las cláusulas when son opcionales. Una alternativa está abierta si B_i es true o se omite el when. Las B_i no pueden referenciar parámetros del entry call, y tampoco hay expresión de scheduling. Demora al proceso hasta que el accept en alguna alternativa abierta pueda ejecutarse (hay una invocación pendiente del entry). Puede contener una alternativa **else** (que se elige si no se puede elegir otra alternativa), **or delay** (se elige si transcurrió el intervalo, como un timeout), **or terminate** (se elige si todas las tareas que hacen RV con esta terminaron o están esperando un terminate).

RPC y rendezvous → un proceso inicia la comunicación con un call, que bloquea al llamador hasta que la operación es servida y se retornan los resultados. Ideales para interacciones C/S, pero difícil programar algoritmos filtros o peers que intercambian información (para éstos es mejor PMA).

Notación de Primitivas Múltiples: combina RPC, Rendezvous y PMA en un paquete coherente.
- Brinda gran poder expresivo combinando ventajas de las 3 componentes, y poder adicional.
- Programas = colecciones de módulos. Una operación visible se declara en la especificación del módulo. Puede ser invocada por procesos de otros módulos, y es servida por un proceso o procedure del módulo que la declara.
- También se usan operaciones locales, que son declaradas, invocadas y servidas dentro del cuerpo de un único módulo.
Una operación puede ser invocada por call sincrónico o por send asincrónico:

call Mname.op(argumentos)
send Mname.op(argumentos)

El call termina cuando la operación fue servida y los resultados fueron retornados. El send termina tan pronto como los argumentos fueron evaluados. Una operación es servida por un procedure (proc) o por rendezvous (sentencias in). La elección la toma el programador del módulo.

Invocación	*Servicio*	*Efecto*
call	**proc**	llamado a procedimiento
call	**in**	rendezvous
send	**proc**	creación dinámica de proceso
send	**in**	PMA

Un llamado a procedure es local si el llamador y el proc están en el mismo módulo; sino, es remoto. Una operación no puede ser servida tanto por proc como por in pues el significado no sería claro. Pero 1 operación puede ser servida por más de una sentencia de E/, quizás en más de un proceso en el módulo que la declara. En este caso, los procesos comparten la cola de invocaciones pendientes.

Clase 10

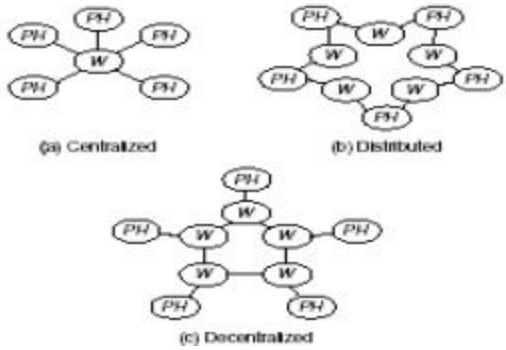
Paradigmas para la interacción entre procesos

Hay 3 esquemas básicos de interacción e/ procesos: productor / consumidor, cliente / servidor e interacción entre pares. Se pueden combinar de muchas maneras, dando lugar a 7 paradigmas o modelos de interacción entre procesos:

Paradigma 1: servidores replicados. Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos. Ej: File servers con múltiples clientes y una instancia de servidor por cliente (o por File, o por Unidad de almacenamiento).
Paradigma 2: algoritmos heartbeat. Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive. Ejemplo: Modelos biológicos / Problema de los N Cuerpos / Modelos de simulación paramétrica.
Paradigma 3: algoritmos pipeline. La información recorre una serie de procesos utilizando alguna forma de receive/send. Se supone una arquitectura de procesos/procesadores donde la salida de uno es entrada del siguiente. Los procesadores pueden distribuirse datos o funciones. Ejemplo: Redes de Filtros, Tratamiento de Imágenes.
Paradigma 4: probes (send) y echoes(receive). La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información. Un ejemplo clásico es recuperar la topología activa de una red móvil o hacer un broadcast desde un nodo cuando no se “alcanzan” o “ven” directamente todos los destinatarios.
Paradigma 5: algoritmos broadcast. Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas. En general en sistemas distribuidos con múltiples procesadores, las comunicaciones colectivas representan un costo crítico en tiempo. Un ejemplo típico es la sincronización de relojes en un Sistema Distribuido de Tiempo Real.

Paradigma 6: token passing. En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. La arquitectura puede ser un anillo, caso en el cual el manejo se asemeja a un pipeline, pero también puede ser cualquier topología (tipo objetos distribuidos). Los tokens normalmente habilitan el control para la toma de decisiones distribuídas.
Paradigma 7: manager/workers. Implementación distribuida del modelo de bag of tasks. Un procesador controla los datos y/o procesos a ejecutarse y múltiples procesadores acceden a él para acceder a datos/procesos y ejecutar las tareas distribuidas.

Paradigma de servidores replicados. Ejemplo: filósofos.
Un server puede ser replicado cuando hay múltiples instancias de un recurso: c/ server maneja una. También puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.



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

Filósofos centralizado (sin deadlock – no fair – cuello de botella)

```
module Mesa
   op tomar_ten(int), liberar_ten (int);
body
   process Mozo {
          bool comiendo[5] =([5] false);
          while (true)
             in tomar_ten(i) and not (comiendo[izq(i)] and
                    not coimiendo[der(i)]  → comiendo[i] = true;
             □ liberar_ten(i) →
                    comiendo[i] = false;
             ni      }
end Mesa
Process Filosofo [i = 0 to 4] {
   while (true) {
          call tomar_ten(i);
          come;
          call liberar_ten(i);
          piensa;
    }
 }
```

Filósofos Distribuido (fair – sin deadlock – no hay cuello de botella – más mensajes)

```
Process Filosofo[i = 0 to 4]  {
     INT primero = i, segundo = i+1;
     IF (i == 4)   { primero=0; segundo=4;}
     WHILE (true)  {
          Call Mozo[primero]. TomarTen( );
          Call Mozo[segundo]. TomarTen( );
          COMER;
          Send Mozo[primero]. LiberarTen ( );
          Send Mozo[segundo].LiberarTen ( );
          PENSAR;          }
}
Module Mozo[5]
    op TomarTen( ), LiberarTen( );
BODY
    Process El_Mozo   {
         WHILE (true) {
              receive TomarTen ( );
              receive LiberarTen ( );       }       }
End Mozo;
```

Filósofos Descentralizado

La solución descentralizada tiene un mozo por filósofo. El algoritmo usado por los mozos es *token passing*, donde tokens = tenedores. La solución puede adaptarse para coordinar acceso a archivos replicados, o dar solución eficiente al problema de EM distribuida. Cada tenedor es un token que tiene uno de los dos mozos o está en tránsito entre ellos. Cuando el filósofo quiere comer, le pide a su mozo que adquiera dos tenedores. Si el mozo no los tiene, interactúa con sus mozos vecinos para obtenerlos. Luego mantiene el control mientras el filósofo come. Debe evitarse el deadlock, que podría darse si un mozo necesita dos tenedores y no los puede lograr.

Cuando su filósofo no está comiendo, el mozo debe poder ceder sus tenedores, pero debe evitar pasar ida y vuelta un tenedor entre mozos sin que sea usado. Para evitar deadlock, el mozo debe ceder un tenedor que ya fue usado, pero debe mantener uno que adquirió y no usó. Cuando un Phil empieza a comer, su mozo marca los tenedores como “sucios”. Cuando otro quiere un tenedor, si está sucio y no se usa, lo limpia y lo cede. El que lo recibe, mantiene el tenedor limpio hasta que sea usado. Un tenedor sucio puede ser reusado hasta que lo necesite otro. (Filósofos “higiénicos”). Inicialmente los tenedores se distribuyen asimétricamente y todos están sucios. La solución evita la inanición.

Algoritmos heartbeat

Paradigma heartbeat: útil para soluciones iterativas que se quieren paralelizar. Usando un esquema “divide & conquer” se distribuye la carga e/ los workers; c/u es responsable de actualizar una parte. Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos. Cada “paso” debiera significar un progreso hacia la solución Ej: grid computations (imágenes o PDE), autómatas celulares (simulación de fenómenos como incendios o crecimiento biológico).

```
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; }
```

}

Algoritmos heartbeat. Labeling de regiones en imágenes

Si tenemos una imagen representada en una matriz Imagen[mxn], en muchos casos es importante separarla en zonas que se identifiquen con un label (ej: conteo de células, reconocimiento tumores). Un modo natural de paralelizar este tipo de aplicaciones es dividir la Imagen en P “strips” o conjunto de filas en las que un proceso trata de poner un label a cada pixel. En c/ ciclo los resultados deben ser transmitidos entre los procesos vecinos porque las “zonas” pueden ser compartidas. Esto establece una especie de “barrera” entre los workers vecinos. Las iteraciones pueden ser fijas o tener un proceso coordinador o manager que detecte cuando ninguno de los P procesos workers encontró cambios en el labeling. El resultado final es la imagen descompuesta en K “regiones” cada una con un label. Luego se puede avanzar en el tratamiento de cada región.

Paradigma de pipelining

Un pipeline es un arreglo lineal de procesos “filtro” que reciben datos de un puerto (canal) de E/ y entregan resultados por un canal de S/.

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.

Paradigma de Prueba-Eco (Probe-Echo)

Prueba-eco se basa en el envío de un msg (“probe”) de un nodo al sucesor, y la espera posterior del “eco” (mensaje de respuesta). Los probes se envían en **paralelo** a todos los sucesores. Este paradigma es el análogo concurrente de DFS Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un n° fijo de nodos activos (ejemplo: redes móviles).

El problema de hacer un broadcast a todos los nodos de una red es clásico en procesamiento distribuido. Un primer enfoque es suponer que algún nodo tiene la topología de la red en alguna forma de matriz donde la entrada [i, j] es true si los nodos están conectados. Resolver el broadcast con un spanning tree supone un nodo (el origen) que tiene decidido el routing y lo envía a sus sucesores, conjuntamente con el mensaje m (actúa como raíz del árbol). C/ nodo a su vez recibe el mensaje m y el spanning tree t y visualiza cuáles son los “hijos” a los que debe reenviar el mensaje. El proceso es asincrónico, pero podría ser sincrónico (menor eficiencia). El broadcast se complica cuando NO se conoce la topología. Algoritmo simétrico: cada nodo al recibir un msg lo reenvía a todos sus vecinos, incluyendo el emisor. Después recibe copias que ignora.

En la mayoría de las LAN los procesadores comparten un canal común (Ethernet o token ring) ⇒ c/ procesador se conecta directamente con los otros. Estas redes normalmente soportan una primitiva especial broadcast que transmite un mensaje de un procesador a todos los otros: broadcast ch(m); Los msgs broadcast de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los msgs enviados por procesos A y B podrían ser recibidos por otros en distinto orden. Se puede usar broadcast para diseminar información (ej: intercambiar info de estado de procesadores en LAN), o p/ resolver problemas de sincronización distribuida (Ej: semáforos

distribuidos). La base es un ordenamiento total de eventos de comunicación mediante el uso de relojes lógicos.

Algoritmos 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 1er 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 pgms 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 BDD. 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).

Manager/Workers: Bag of Tasks Distribuido

El concepto de bag of tasks usando VC supone que un conjunto de workers comparten una "bolsa" con tareas o procesos 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.

En la implementación de este paradigma con mensajes en lugar de MC, 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.

<u>Programación Paralela</u>

Programa Concurrente: múltiples procesos.

Programa Distribuido: pgm concurrente en el cual los procesos se comunican y sincronizan por PM, RPC o Rendezvous.

Programa Paralelo: pgm concurrente escrito para resolver un problema en menos tiempo que el secuencial. El objetivo ppal es reducir el tiempo de ejecución, o resolver problemas + grandes o con > precisión en el mismo tiempo. Un programa paralelo puede escribirse usando VC o PM. La elección la dicta el tipo de arquitectura.

Los dos modos tradicionales del descubrimiento científico son teoría y experimentación. El 3er modo es la modelización computacional, que usa computadoras para simular fenómenos y tratar cuestiones del tipo "what if?" Entre las diferentes aplicaciones de cómputo científicas y modelos computacionales existen tres técnicas fundamentales:

(1) Computación de grillas (soluciones numéricas a PDE, imágenes). Dividen una región espacial en un conjunto de puntos.

(2) Computación de partículas (modelos que simulan interacciones de partículas individuales como moléculas u objetos estelares).

(3) Computación de matrices (sistemas de ecuaciones simultáneas).

Problemas “grand challenge”. Abarcan química cuántica, mecánica estadística, cosmología y astrofísica, dinámica de fluidos computacional, diseño de materiales, biología, farmacología, secuencia genómica, ingeniería genética, medicina y modelización de órganos y huesos humanos, pronóstico del tiempo, sensado remoto, física de partículas etc.

Diseño de algoritmos paralelos: es necesaria la creatividad, la mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes. Consta de 4 etapas:
- particionamiento (descomposición en tareas).
- comunicación (estructura necesaria para coordinar la ejecución).
- aglomeración (evaluación de tareas y estructura con respecto a performance y costo, combinando tareas para mejorar).
- mapeo (asignación de tareas a procesadores).

Métricas del paralelismo

Tamaño del problema (W)

Función del tamaño de la entrada. Está dado por el número de operaciones básicas necesarias para resolver el problema en el algoritmo secuencial más rápido. Es incorrecto pensar, por ej, que en problemas con matrices de nxn el tamaño de problema es n pues la interpretación cambiaría de un problema a otro. Por ej, duplicar el tamaño de la E/ resulta en un incremento de 8 veces en el tiempo de ejecución serial p/ la multiplicación y de 4 veces p/ la suma. El tiempo de ejecución paralelo, para un sistema paralelo dado, es función del tamaño del problema y el n° de procesadores (Tp(W,p)).

Speedup (S)

S es el cociente entre el tiempo de ejecución serial del algoritmo serial conocido más rápido (T_s) y el tiempo de ejecución paralelo del algoritmo elegido (T_p)

$$S = T_s / T_p$$

Las diferentes definiciones de tiempo de ejecución serial llevan a distintas definiciones de speedup. Rango de valores: en general entre 0 y p La gráfica (curva de speedup) refleja p en las abcisas y S en las ordenadas Speedup lineal o perfecto, sublineal y superlineal.

Eficiencia (E)

Cociente entre speedup y número de procesadores

$$E = T_s / pT_p$$

Mide la fracción de tiempo en que los procesadores son útiles para el cómputo El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.
No debieran usarse speedup y eficiencia como métricas independientes del tiempo de corrida.

Factores que limitan el Speedup (Overhead paralelo)

-entrada/salida (el % es alto respecto de la computación)
- algoritmo (no adecuado, necesidad de rediseñar)
- excesiva contencíón de memoria (es necesario rediseñar el código p/tener 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)
- alto porcentaje de código secuencial (Amdahl)
- overhead paralelo (ciclos adicionales de CPU para crear procesos, sincronizar, etc)

Función de overhead: $T_o(W,p) = pT_p - W$
Suma todos los overheads en que incurren todos los procesadores debido al paralelismo.

Costo
El costo de un sistema paralelo es el producto de T_p y p. Refleja la suma del tiempo que cada procesador utiliza en la resolución del problema Puede expresarse la eficiencia como el cociente entre el tiempo de ejecución del algoritmo secuencial conocido más rápido y el costo de resolver el problema en p procesadores También suele referirse como trabajo o producto procesador-tiempo.

Grado de concurrencia o paralelismo
C(W) es el número máximo de tareas que pueden ejecutarse simultáneamente en cualquier momento del algoritmo paralelo Para un W dado, el algoritmo paralelo no puede usar más de C(W) procesadores. C(W) depende sólo del algoritmo, no de la arquit. Es una función discreta del tiempo. La gráfica de C(W) vs. tiempo es el perfil de concurrencia. Supone un número ilimitado de procesadores y otros recursos, lo que no siempre es posible de tener.

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