Tópicos de programación Concurrente y Paralela

Facultad de Informática UNLP

UNLu

Clase 2 - Conceptos generales, desafíos y escenarios de resolución

Aspectos de Programación Secuencial

- Cada proceso concurrente es un programa secuencial $\Rightarrow$ es necesario referirse a algunos aspectos de la programación secuencial

- La Programación Secuencial estructurada puede expresarse con tres clases de instrucciones básicas: asignación, alternativa (decisión) e iteración (repetición con condición).

- Es necesaria alguna instrucción para expresar la concurrencia…

Paradigmas de resolución de programas concurrentes

- Si bien el número de aplicaciones es muy grande, en general los “patrones” de resolución concurrentes son pocos:

1- Paralelismo iterativo

2- Paralelismo recursivo

3- Productores y consumidores (*pipelines* o *workflows*)

4- Clientes y servidores

5- Pares que interactúan (*interacting peers*)

Paradigmas de resolución de programas concurrentes

- En el **paralelismo iterativo** un programa consta de un conjunto de procesos (posiblemente idénticos) cada uno de los cuales tiene 1 o más loops $\Rightarrow$ cada proceso es un programa iterativo.

- Los procesos cooperan para resolver un único problema (por ejemplo un sistema de ecuaciones), pueden trabajar independientemente, y comunicarse y sincronizar por memoria compartida o memoria distribuida.

- Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones (división por filas, división por columnas, strips, etc.).

Paradigmas de resolución de programas concurrentes

- En el **paralelismo recursivo** el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (Dividir y conquistar)

- Ejemplos clásicos son el sorting by merging, el cálculo de raíces en funciones continuas, problema del viajante, juegos (tipo ajedrez)

Paradigmas de resolución de programas concurrentes

- Los esquemas **productor-consumidor** 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.
- Ejemplos a distintos niveles de Sistema Operativo.

Paradigmas de resolución de programas concurrentes

- **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 (rendezvous y RPC por ejemplo en memoria distribuída, monitores por ejemplo en memoria compartida).
- El soporte distribuido puede ser simple (LAN) o extendido a la WEB.

Paradigmas de resolución de programas concurrentes

- En los esquemas de **pares que interactúan** los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea y completar el objetivo.

- Permite mayor grado de asincronismo que Cliente/Servidor

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

Acciones Atómicas y Sincronización

- Estado de un Programa concurrente
- Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
- Una **acción atómica** hace una transformación de estado indivisibles (estados intermedios invisibles para otros procesos).
- Ejecución de un programa concurrente → **intercalado (interleaving)** de las acciones atómicas ejecutadas por procesos individuales.
- Interacción → no todos los interleavings son aceptables.
- **Historia** de un programa concurrente (trace).

Acciones atómicas y Sincronización

```
int buffer;
PROCESS P1() {
    int x;
    while (true) {
        read (x);  // p11
        buffer = x; // p12
    }
}
```

```
PROCESS P2() {
    int y;
    while (true) {
        y = buffer; // p21
        print (y);  // p22
    }
}
```

Posibles interleavings (historias):

p11, p12, p21, p22, p11, p12, p21, p22, ... ☑

p11, p12, p21, p11, p22, p12, p21, p11, p12, p22, ... ☑

p11, p21, p12, p22, ... ☒

p21, p11, p12, ... ☒

Hay algunas historias que no son válidas…

Acciones atómicas y Sincronización

Sincronizar ⇒ Combinar acciones atómicas de grano fino (fine-grained) en acciones (compuestas) de grano grueso (coarse grained) que den la exclusión mutua.

Sincronizar ⇒ 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

Acciones atómicas y Sincronización. Atomicidad de grano fino

Una acción atómica de grano fino se debe implementar por hardware

La operación de asignación A=A+1 es atómica?

NO $\Rightarrow$ Es necesario cargar el valor de A en un registro, sumarle 1 y volver a cargar el valor del registro en A

Qué sucede con algo del tipo X=X+X?

$\Rightarrow$ (i) Load PosMemX, Acumulador
(ii) Add PosMemX, Acumulador
(iii) Store Acumulador, PosMemX

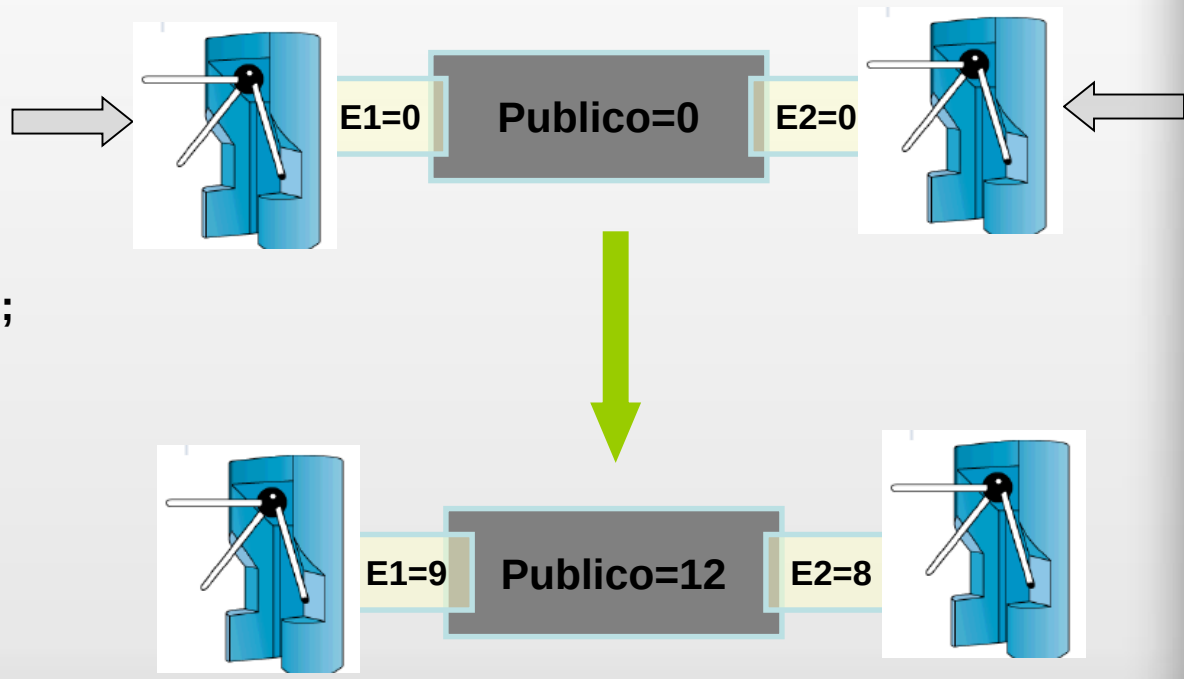
Acciones atómicas y Sincronización. El problema de la *interferencia*

Interferencia: 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?

```
int Publico, E1, E2;
PROCESS P1(){
    while (true) {
        //esperar llegada
        E1 = E1 + 1;
        Publico = Publico + 1;
    }
}

PROCESS P2(){
    while (true) {
        //esperar llegada
        E2 = E2 + 1;
        Publico = Publico + 1;
    }
}
```



Acciones atómicas y Sincronización. Atomicidad de grano fino

La lectura y escritura de las variables x,y,z son atómicas.

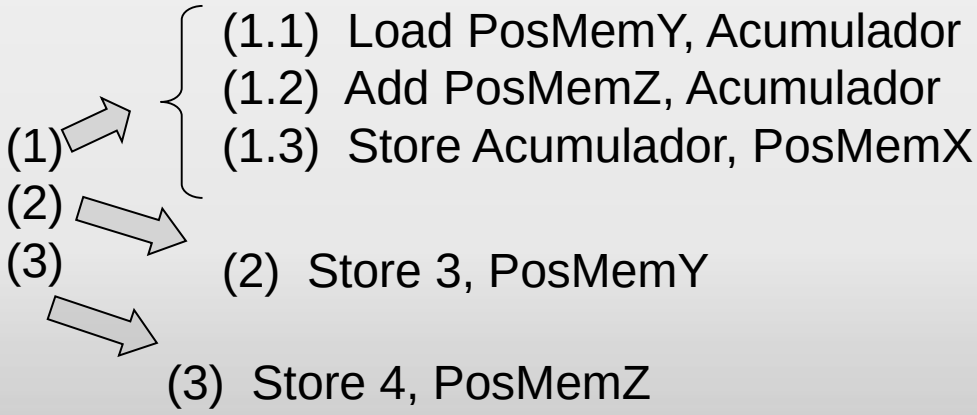
int x = 0, y = 4, z = 2;

```
PROCESS P1(){
    x = y + z; // (1)
}
```

```
PROCESS P2(){
    y = 3; // (2)
}
```

```
PROCESS P3() {
    z = 4; // (3)
}
```

⇒ Cuáles son los posibles resultados con hasta 3 procesadores (no necesariamente de igual velocidad) ejecutando los procesos



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)

Acciones atómicas y Sincronización. Atomicidad de grano fino

"Interleaving extremo"
(Ben-Ari & Burns)

Dos procesos que realizan (cada uno) *k* iteraciones de la sentencia N=N+1 (N compartida init 0)

```
int N;
PROCESS P1() {
      int i;
      for (i = 0; i < K; i++)
            N = N + 1;
}
```

```
PROCESS P2() {
      int i;
      for (i = 0; i < K; i++)
            N = N + 1;
}
```

Cuál puede ser el valor final de N?

- 2K
- entre K+1 y 2K-1
- K
- <K (incluso 2…)

Acciones atómicas y Sincronización. Atomicidad de grano fino

Cuándo valdrá k?

1. Proceso 1: Load N
2. Proceso 2: Load N
3. Proceso 1: Incrementa su copia
4. Proceso 2: Incrementa su copia
5. Proceso 1: Store N
6. Proceso 2: Store N

Acciones atómicas y Sincronización. Atomicidad de grano fino

- En lo que sigue, supondremos máquinas con las siguientes características:
- ***Los valores de los tipos básicos se almacenan en elementos de memoria leídos y escritos como acciones atómicas***
- ***Los valores se cargan en registros, se opera sobre ellos, y luego se almacenan los resultados en memoria***
- ***Cada proceso tiene su propio conjunto de registros (conjuntos disjuntos o context switching)***
- ***Todo resultado intermedio de evaluar una expresión compleja se almacena en registros o en memoria privada del proceso***

Acciones atómicas y Sincronización. Atomicidad de grano fino

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

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

Pero ... normalmente los programas concurrentes no son disjuntos
⇒ Es necesario establecer algún requerimiento más débil ...

Acciones atómicas y Sincronización. Propiedad de *"A lo sumo una vez"*

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

Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

*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 referencia crítica)

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.

⇒ Mecanismo de sincronización para construir una acción atómica *de* grano grueso (*coarse grained*) como secuencia de acciones atómicas de grano fino (*fine grained*) que aparecen como indivisibles

Ej: dos valores que en todo momento deben ser iguales

Ej: Productor y consumidor con una lista enlazada

Especificación de la sincronización

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.

Ej: **await (s>0, s = s – 1;);**

Sentencia con alto poder expresivo, pero el costo de implementación de la forma general de await (Exclusión Mutua y Sincronización por Condición) es alto

Especificación de la sincronización

Sólo exclusión mutua ⇒ EM(S);
Ej: EM(x = x + 1; y = y + 1;);
El estado interno en el cual x e y son incrementadas resulta invisible a otros procesos que las referencian

Sólo sincronización por condición ⇒ await (B, SKIP);
Ej: await (count > 0, SKIP);

Acciones atómicas incondicionales y condicionales

Especificación de la sincronización

Ejemplo: productor/consumidor con buffer de tamaño N.

```
int cant = 0;
cola Buffer;

PROCESS Productor() {
Telemento elemento;
 while (true) {
        // produce elemento
        await  (cant < N, push(buffer, elemento); cant++;);
  }
 }

PROCESS Consumidor(){
Telemento elemento;
 while (true) {
        await  (cant > 0, pop(buffer, elemento); cant--);
        // consume elemento
  }
}
```

Seguridad y vida

Son dos aspectos complementarios de la *corrección*

seguridad **(safety)**

- nada malo le ocurre a un objeto: asegura estados consistentes
- una *falla de seguridad* indica que algo anda mal

vida **(liveness)**

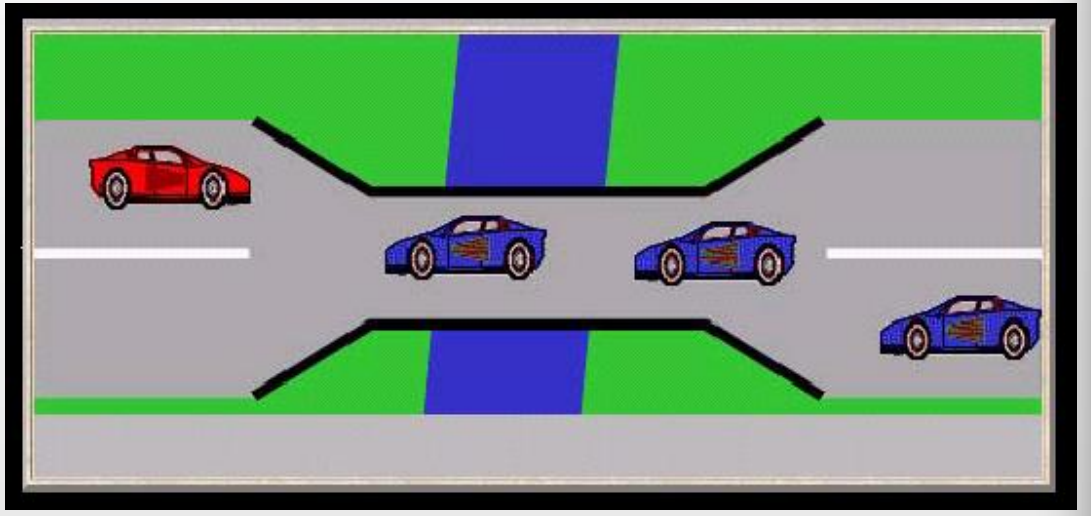
- eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks
- una *falla de vida* indica que se deja de ejecutar

Seguridad y vida

Ejemplo: Puente de una sola vía

Puente sobre río con ancho sólo para una fila de tráfico $\Rightarrow$ los autos pueden moverse concurrentemente si van *en la misma dirección*

- Violación de *seguridad* si dos autos en distintas direcciones entran al puente al mismo tiempo



- Vida: cada auto tendrá *eventualmente* oportunidad de cruzar el puente?

Los temas de seguridad deben balancearse con los de vida

Vida: falta permanente de progreso

- **deadlock**: dependencias circulares
- **señales perdidas**: proceso esperando una notificación para despertarse
- **anidamiento de bloqueos**: un proceso mantiene un bloqueo que otro necesita para despertarse
- **livelock**: una acción que se reintenta y falla continuamente
- **inanición**: el procesador nunca es alocado a un proceso
- **agotamiento de recursos**: un proceso necesita más recursos compartidos para continuar y no hay más
- **falla distribuida**: espera por una E/ desde una máquina remota que cayó

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

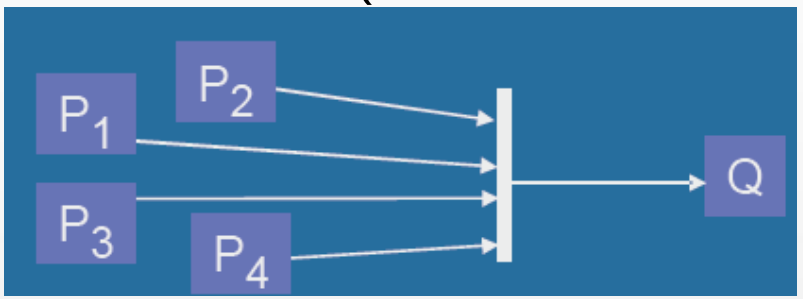
Política: asignar un procesador a un proceso hasta que termina o se demora. ¿Qué sucede en este caso?

```
bool continue = true;

PROCESS P1(){                PROCESS P2(){
    while (continue);            continue = false;
}                            }
```

Tipos de Problemas Básicos de Concurrencia

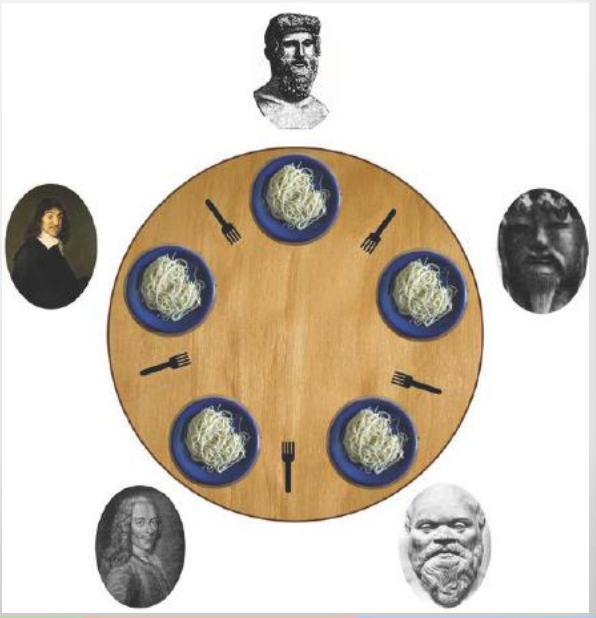
***Exclusión mutua**:* problema de la sección crítica. (Administración de recursos).



Barreras: punto de sincronización

Comunicación

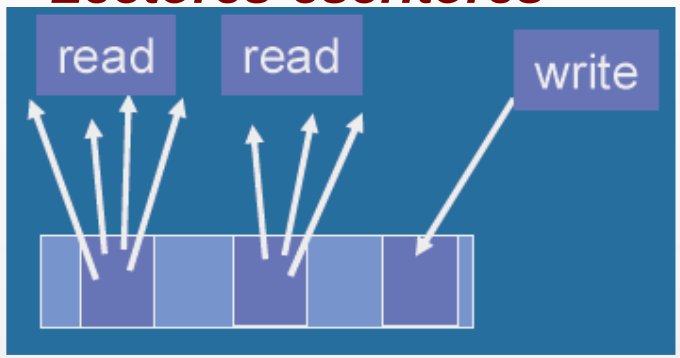
Filósofos: Dijkstra, 1971.
Sincronización multiproceso.
Evitar deadlock e inanición.
Exclusión mutua selectiva



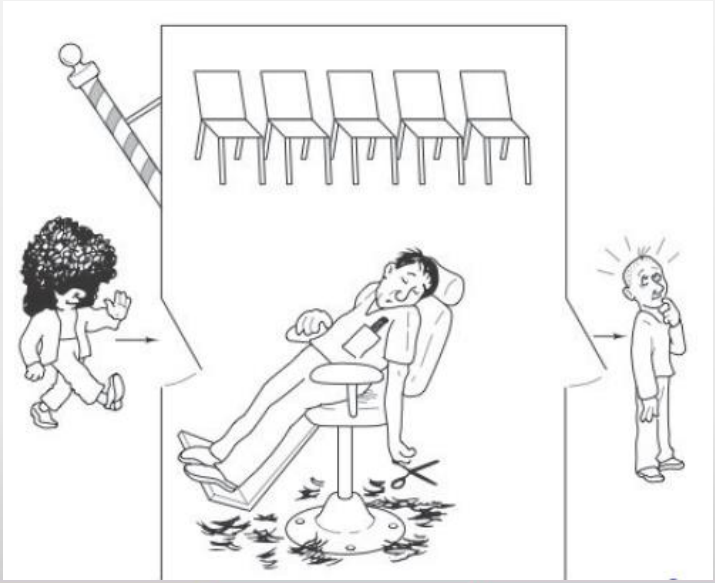
Tipos de Problemas Básicos de Concurrencia

Productor-consumidor

Lectores-escritores



***Sleeping barber**:* Dijkstra.
Sincronización – rendezvous.



Volviendo al hardware ...

... podemos identificar diferentes enfoques para clasificar las arquitecturas paralelas:

- por la organización del espacio de direcciones (memoria compartida / memoria distribuida)

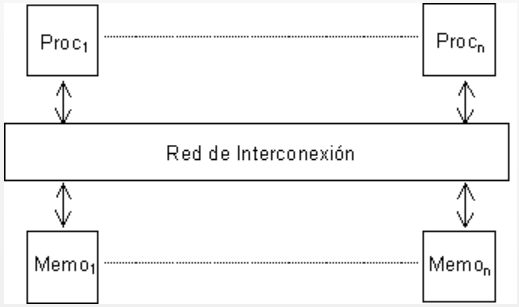
- por el mecanismo de control

- por la granularidad

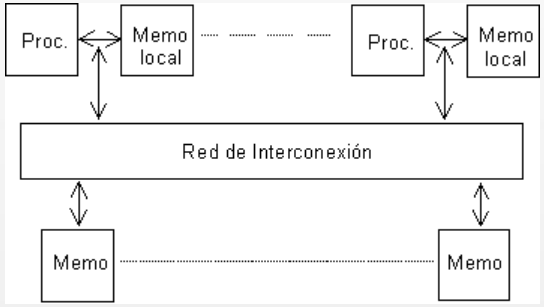
- por la red de interconexión

Clasificación por la organización del espacio de direcciones

- **Multiprocesadores de memoria compartida.**
  - Interacción modificando datos en la memoria compartida.
  - Problema de consistencia.



Esquema UMA



Esquema NUMA

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

Clasificación por mecanismo de control

Propuesta por Flynn (“Some computer organizations and their effectiveness”, 1972).

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

$\Rightarrow$ 4 clases:

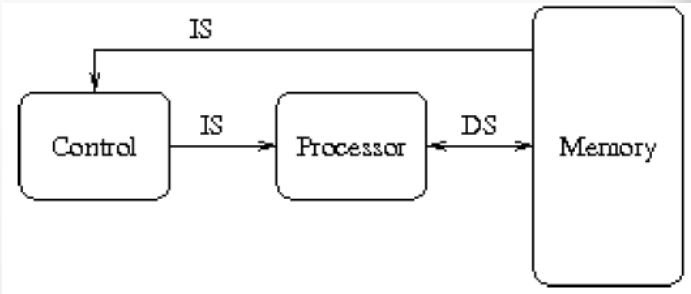
- SISD (Single Instruction Single Data)
- SIMD (Single Instruction Multiple Data)
- MISD (Multiple Instruction Single Data)
- MIMD (Multiple Instruction Multiple Data)

Clasificación por mecanismo de control

SISD: Single Instruction Single Data

Instrucciones ejecutadas en secuencia, una x ciclo de instrucción.

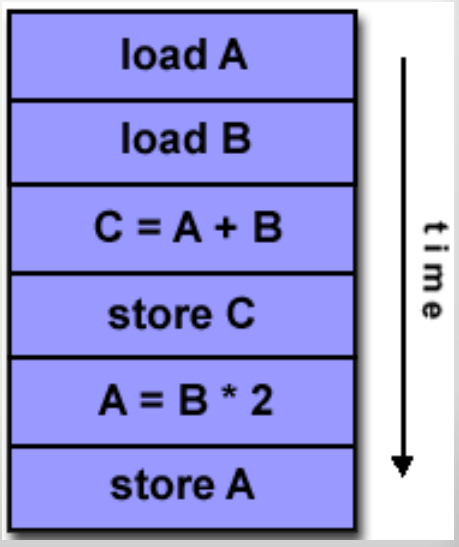
La memoria afectada es usada sólo por esta instrucción



Usada por la mayoría de los uniprocesadores.

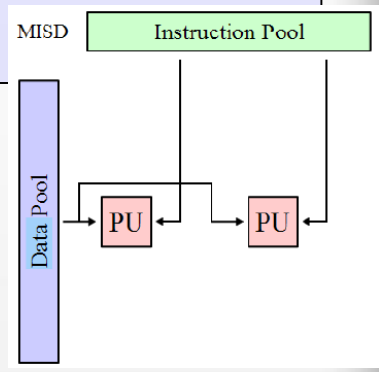
La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memo recibe y almacena datos en las escrituras, y brinda datos en las lecturas.

Ejecución *determinística*



Clasificación por mecanismo de control

MISD: Multiple Instruction Single Data



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

Operación sincrónica (en *lockstep*)

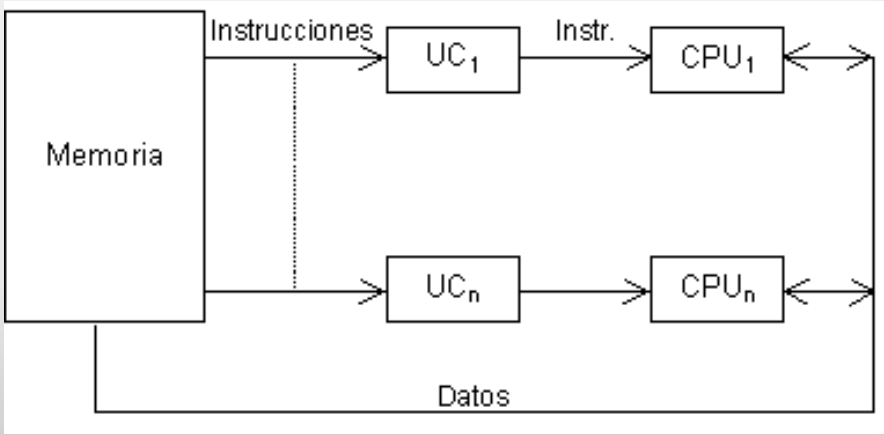
No son máquinas de propósito general (“hipotéticas”, Duncan)

Ejemplos posibles:

- tolerancia a fallas

- múltiples filtros de frecuencia operando sobre una única señal

- múltiples algoritmos de criptografía intentando crackear un único mensaje codificado



Clasificación por mecanismo de control

SIMD: Single Instruction Multiple Data

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

Los procesadores en gral son muy simples.

El host hace broadcast de la instr. Ejecución sincrónica y determinística.

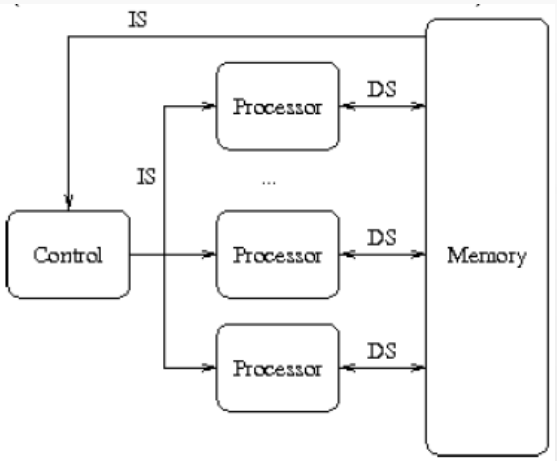
Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones

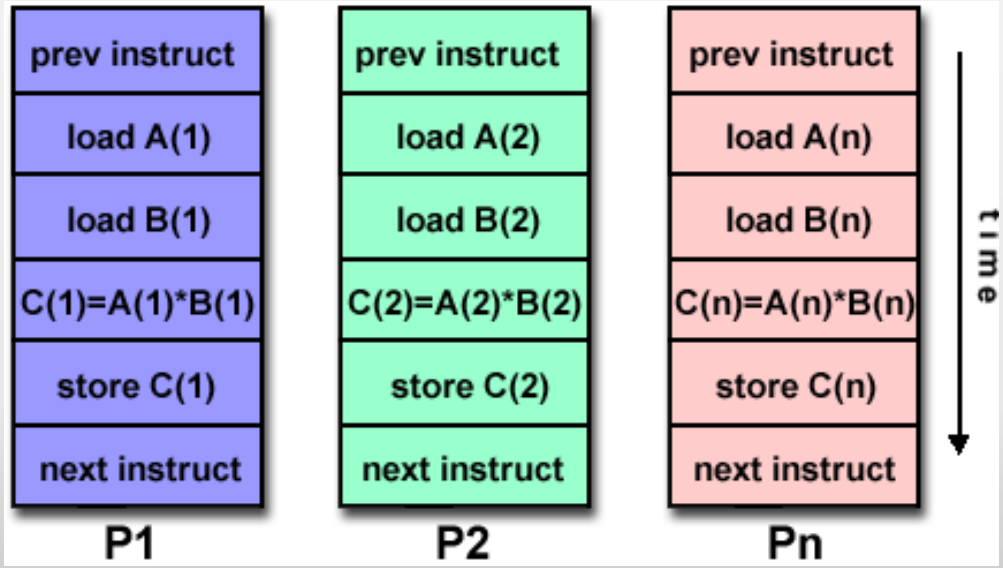
Adecuados para aplicaciones con alto grado de regularidad, (x ej. procesamiento de imágenes).

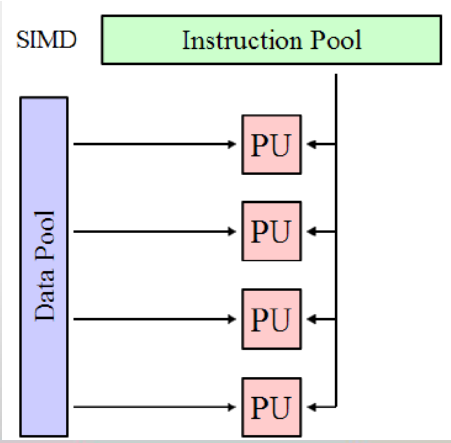
Clasificación por mecanismo de control

SIMD: Single Instruction Multiple Data

Ej: Array Processors. CM-2, Maspar MP-1 y 2, Stream processors en GPU

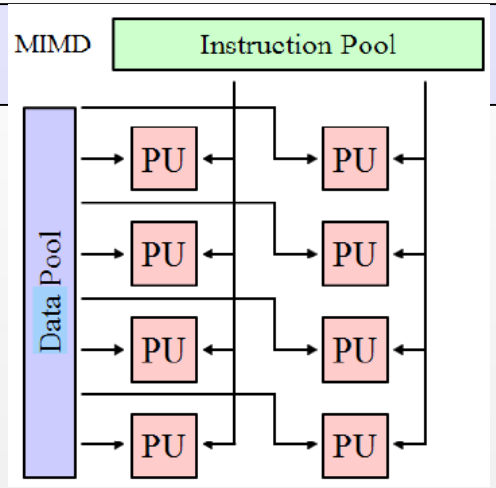






Clasificación por mecanismo de control

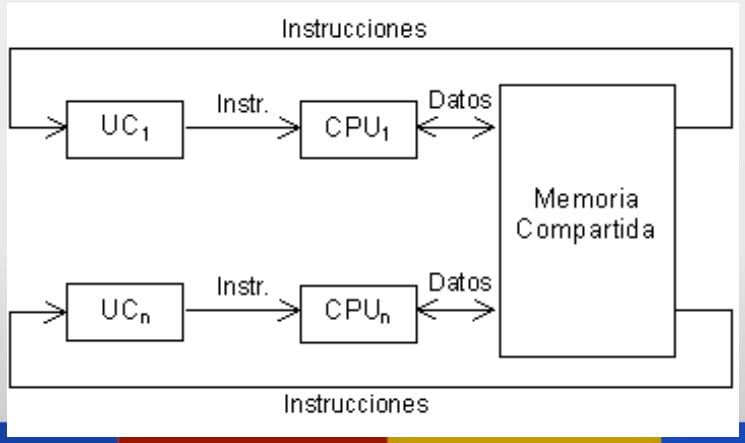
MIMD: Multiple Instruction Multiple Data

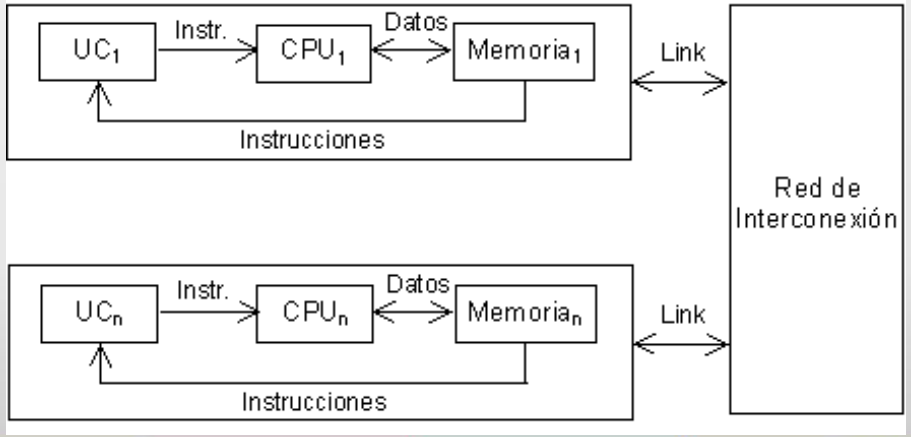


C/ procesador tiene su propio flujo de instrucciones y de datos

⇒ **c/u ejecuta su propio programa**

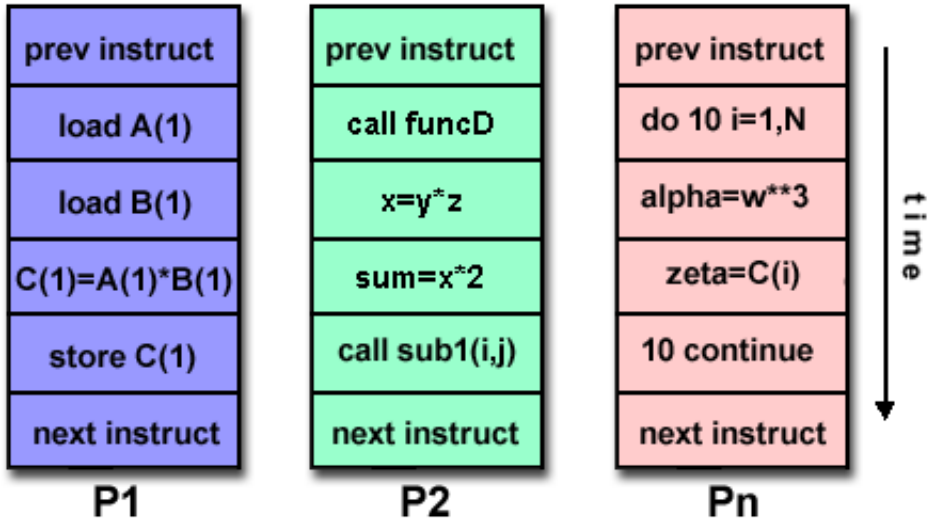
Pueden ser con memoria compartida o distribuida





Clasificación por mecanismo de control

MIMD: Multiple Instruction Multiple Data



Ejemplos: nCube 2, iPSC, CM-5, Paragon XP/S, máquinas DataFlow, red de transputers.

Sub-clasificación de MIMD:

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

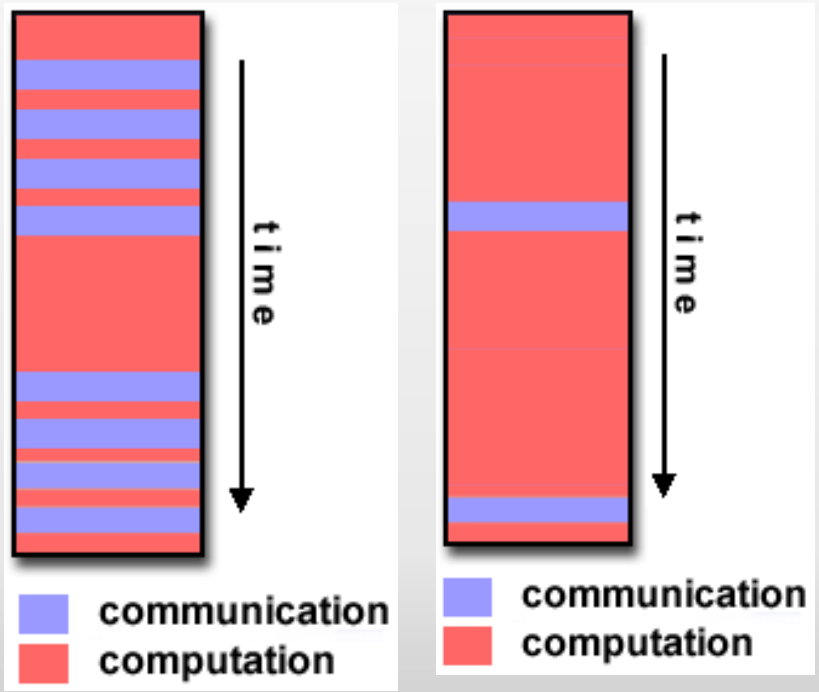
Clasificación por la granularidad de los procesadores

Granularidad

Relación entre el número de procesadores y el tamaño de memoria total.

Grano fino y grano grueso

Puede verse también como la relación entre cómputo y comunicación



Sincronización por variables compartidas. Locks y barreras

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

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

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.

problema de la Sección Crítica

```
PROCESS SC() {       // N procesos
    while (true) {
        // protocolo de entrada;
        sección crítica;
        // protocolo de salida;
        sección no crítica;
    }
}
```

Las soluciones a este problema pueden usarse para implementar sentencias *await* arbitrarias.

Qué propiedades deben satisfacer los protocolos de entrada y salida?

El problema de la Sección Crítica

Exclusión mutua. A lo sumo un proceso está en su SC

***Ausencia de Deadlock (Livelock)*:** Si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.

***Ausencia de Demora Innecesaria*:** Si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.

***Eventual Entrada*:** Un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

Las 3 primeras son propiedades de seguridad, y la 4° de vida.

Solución trivial: hacer atómica la **Sección Crítica**. Pero, ¿cómo hacerlo?

Sincronización Barrier

Problemas resueltos por algoritmos iterativos que computan sucesivas mejores aproximaciones a una respuesta, y terminan al encontrarla o al converger.

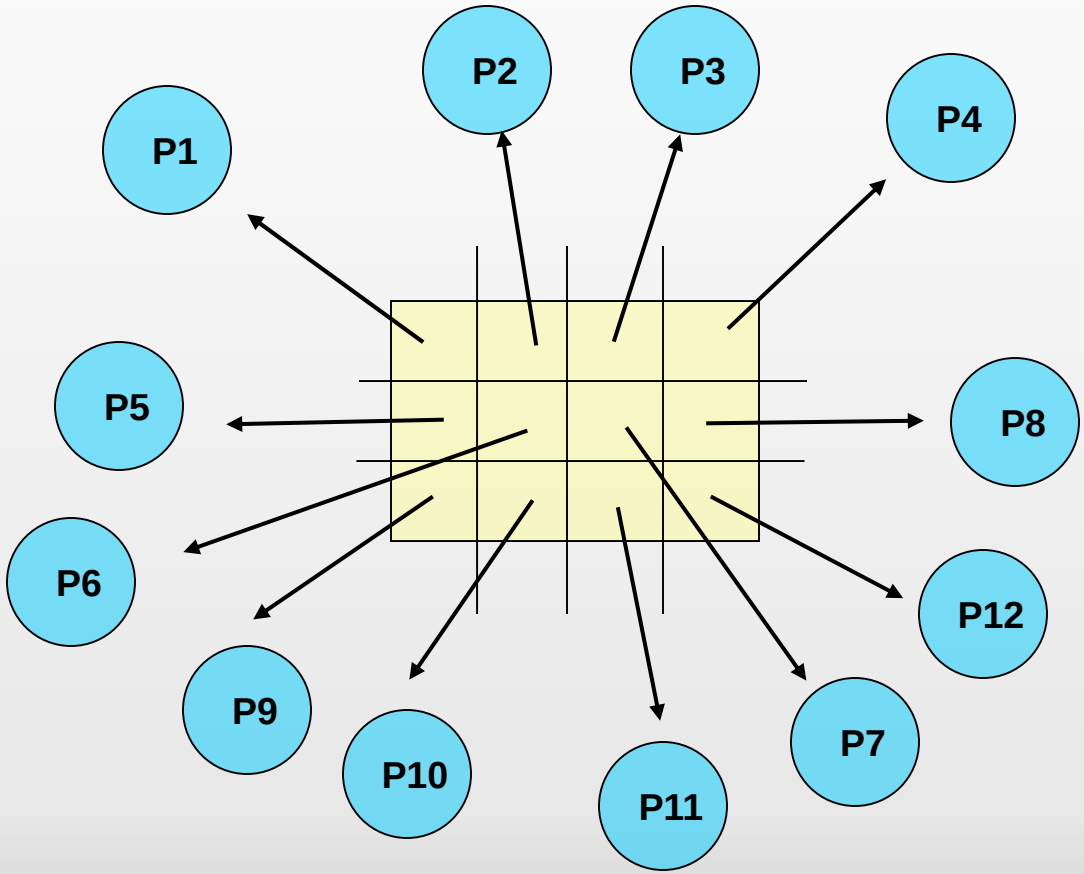
En general manipulan un arreglo, y cada iteración realiza la misma computación sobre todos los elementos del arreglo.

$\Rightarrow$ 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.

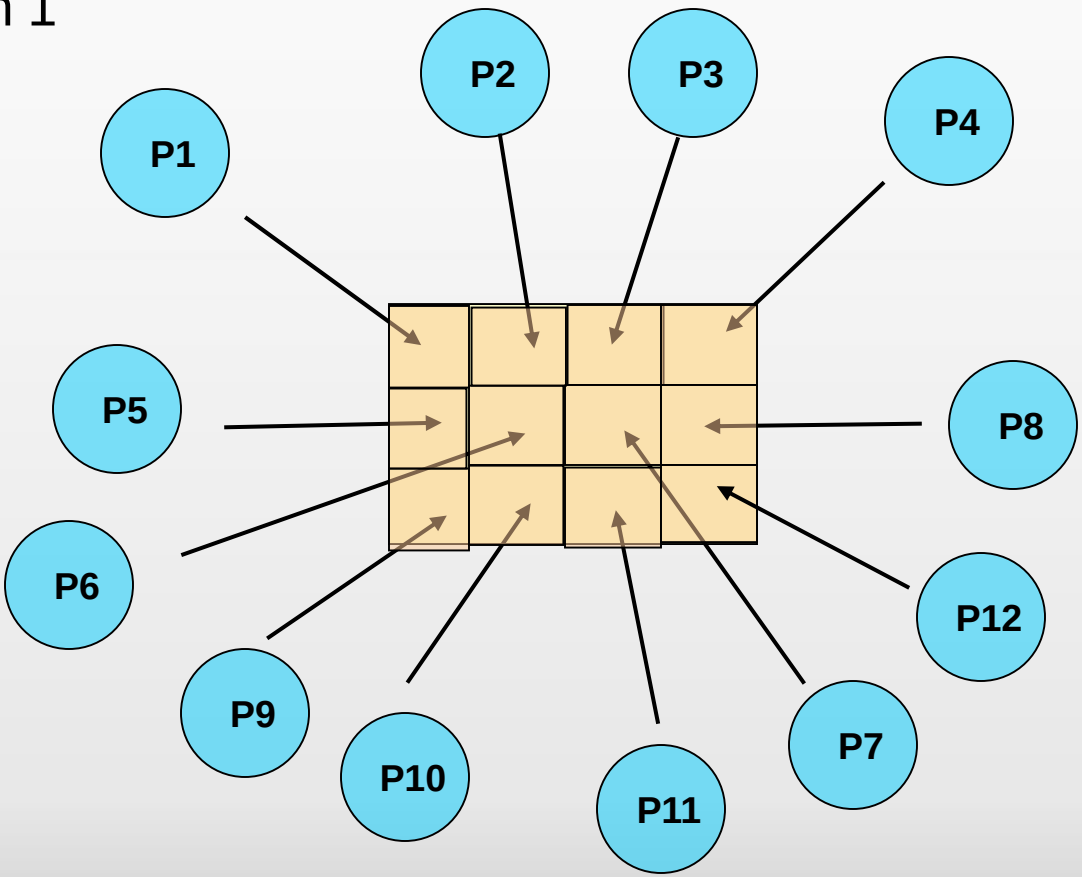
Sincronización Barrier

Inicial



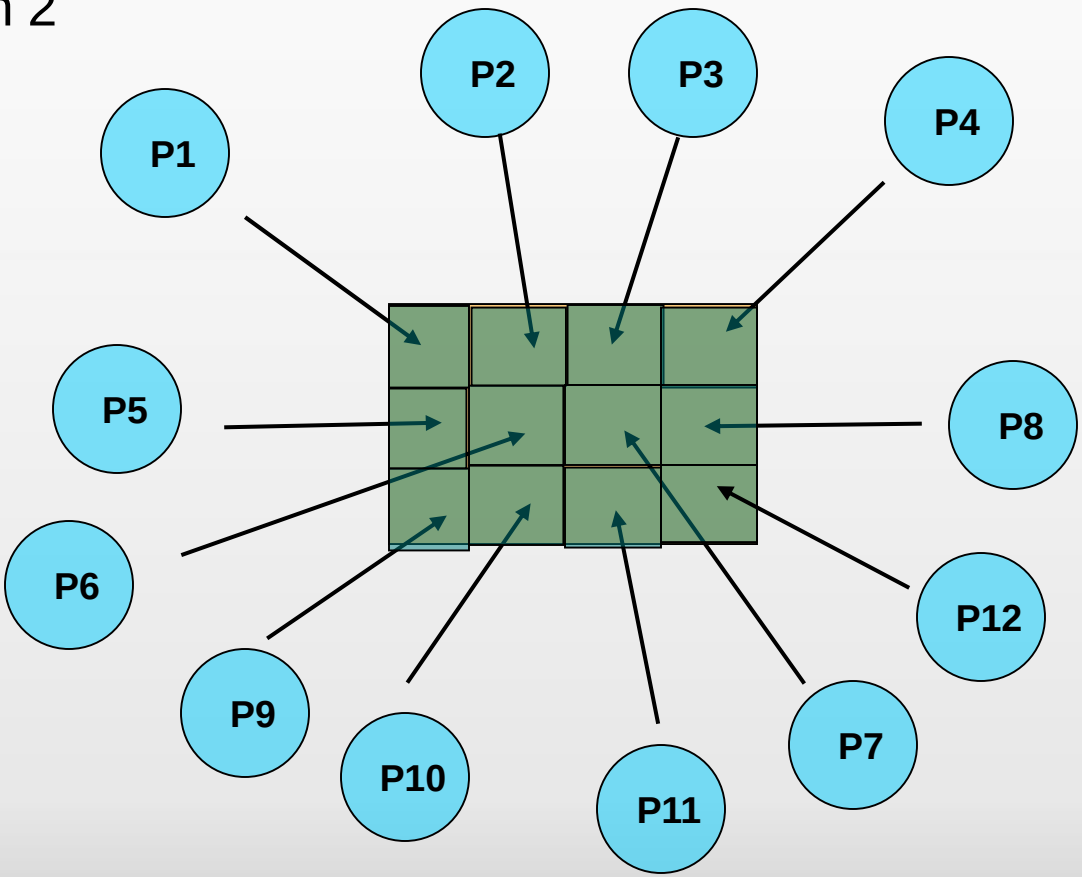
Sincronización Barrier

Iteración 1



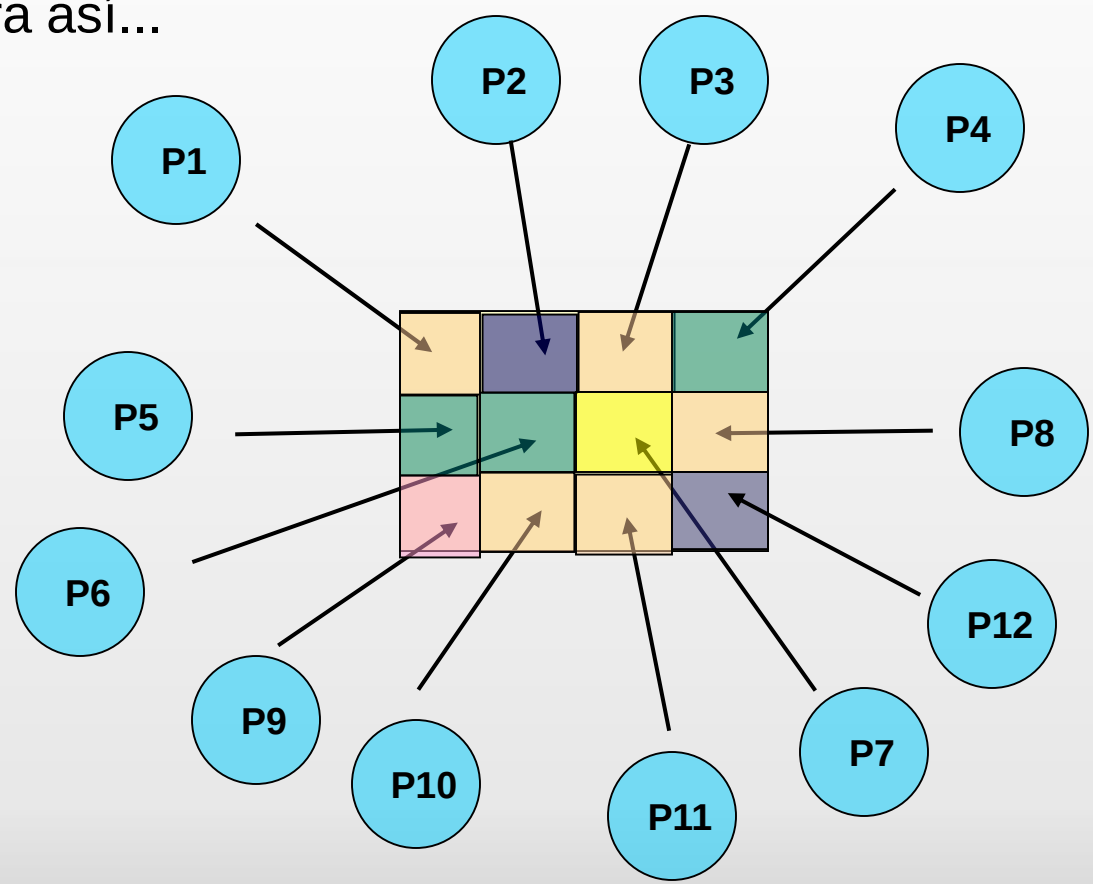
Sincronización Barrier

Iteración 2



Sincronización Barrier

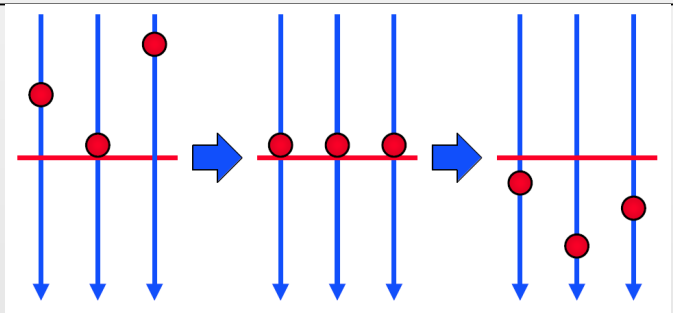
Si no fuera así...



Sincronización Barrier

Crear procesos al comienzo y sincronizarlos al final de cada iteración

```
PROCESS worker() {        // N procesos
   i = getId();
   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 cada iteración es una barrera a la que deben llegar todos antes de permitirles pasar

Defectos de la sincronización por busy waiting

***Protocolos "busy-waiting"*: complejos y sin clara separación entre variables de sincronización y las usadas para computar resultados.**

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

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

⇒ Necesidad de *herramientas* para diseñar protocolos de sincronización.