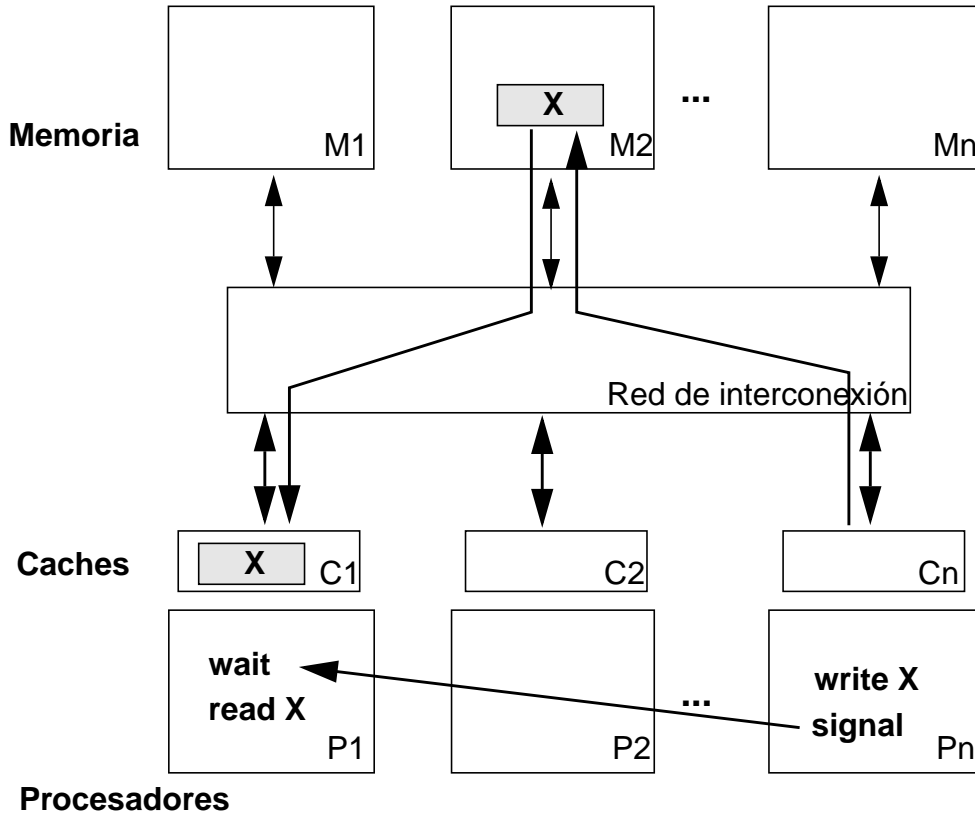


SMP: Symmetric Multiprocessors

- Características Generales**
- Redes de Interconexión**
- Memorias Cache para SMP**
- Sincronización**



Características Generales



- ✓ La red de interconexión debe permitir a cualquier procesador acceder a cualquier posición de memoria y resolver los conflictos en los accesos
- ✓ El sistema de memorias cache permite reducir el tiempo de acceso a los datos y el tráfico de la red
- ✓ Las primitivas de sincronización permiten ordenar los accesos de procesadores diferentes

Características Generales

□ Aspectos Esenciales



Red de Interconexión



Memorias Caches



Sincronización



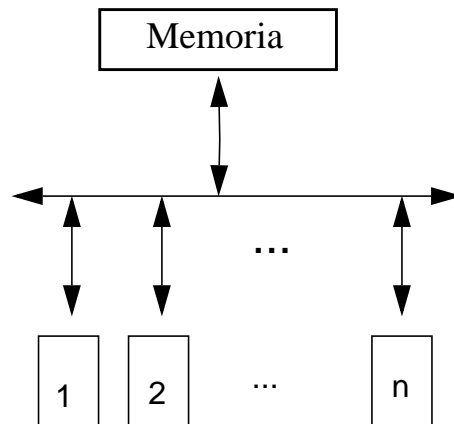
CEPBA



D A C

Redes de Interconexión

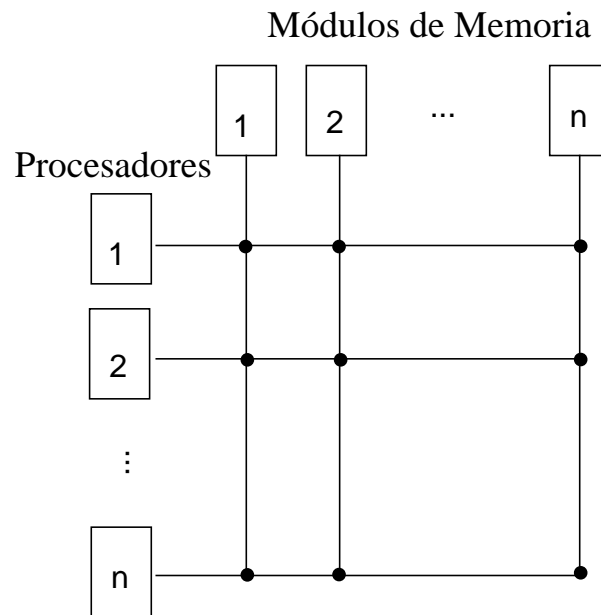
❑ Bus Común



- ✓ Barato
- ✓ Ancho de banda bajo (para un elevado número de procesadores)

Redes de Interconexión

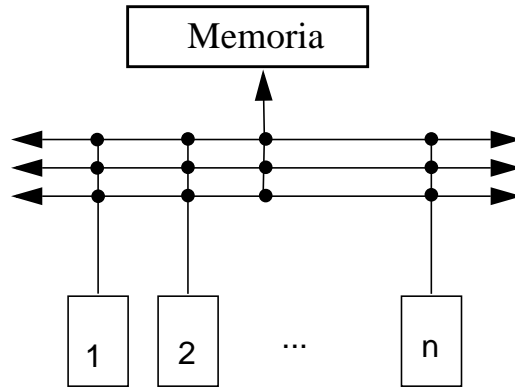
❑ Crossbar



- ✓ Alto ancho de banda
- ✓ Caro (para un número elevado de procesadores)

Redes de Interconexión

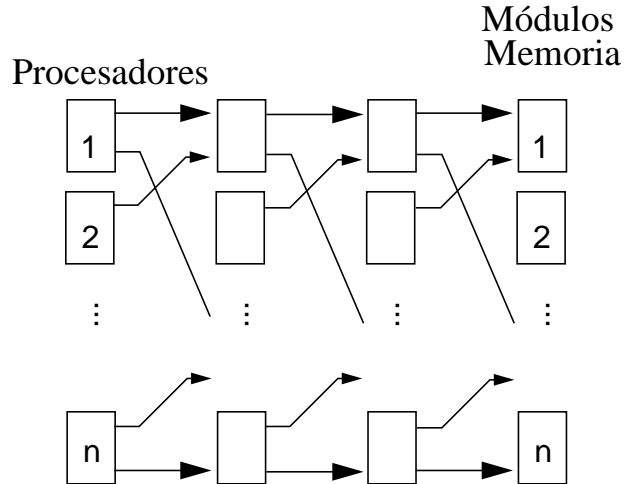
❑ Buses Múltiples



✓ Compromiso entre bus común y crossbar

Redes de Interconexión

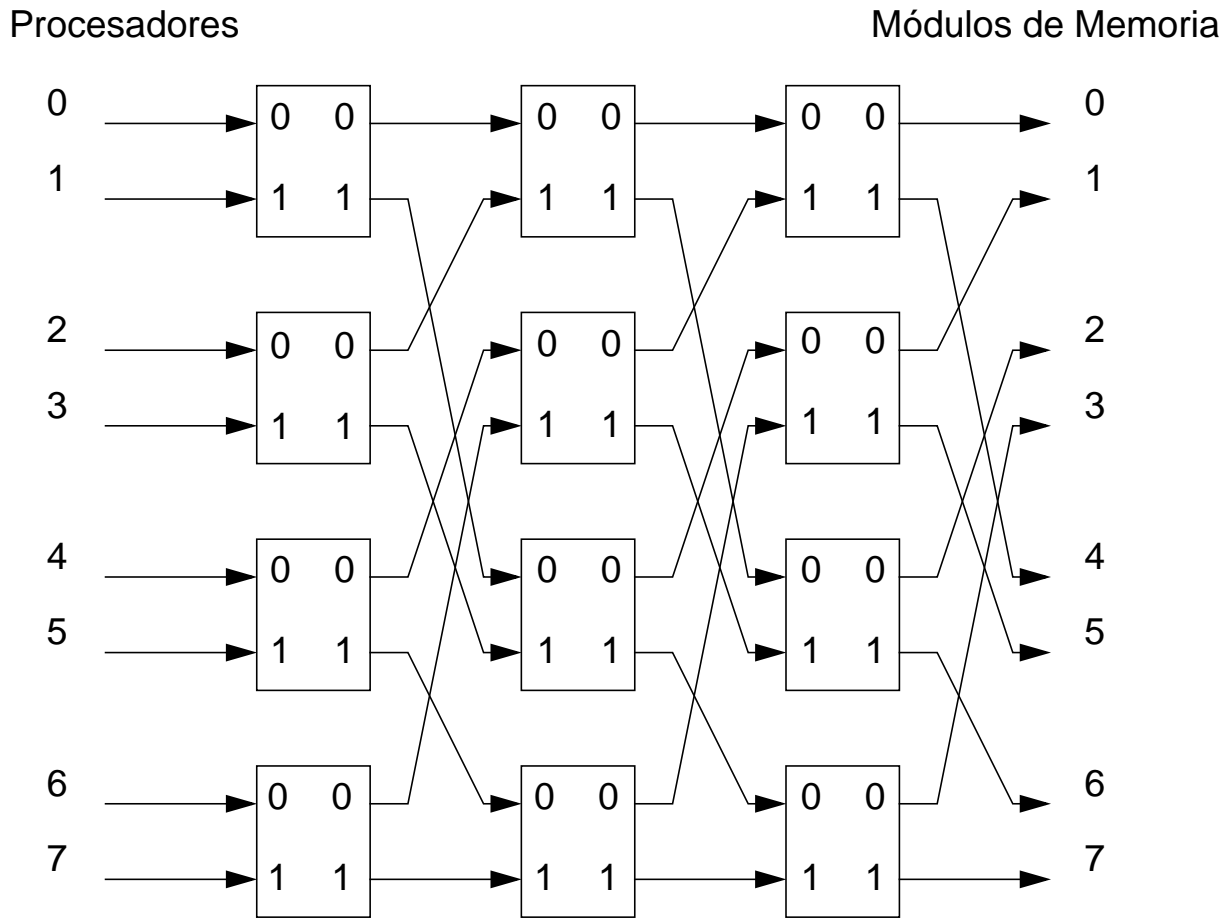
❑ Redes Multietapa



- ✓ Alternativa para un número relativamente elevado de procesadores (256, 512)

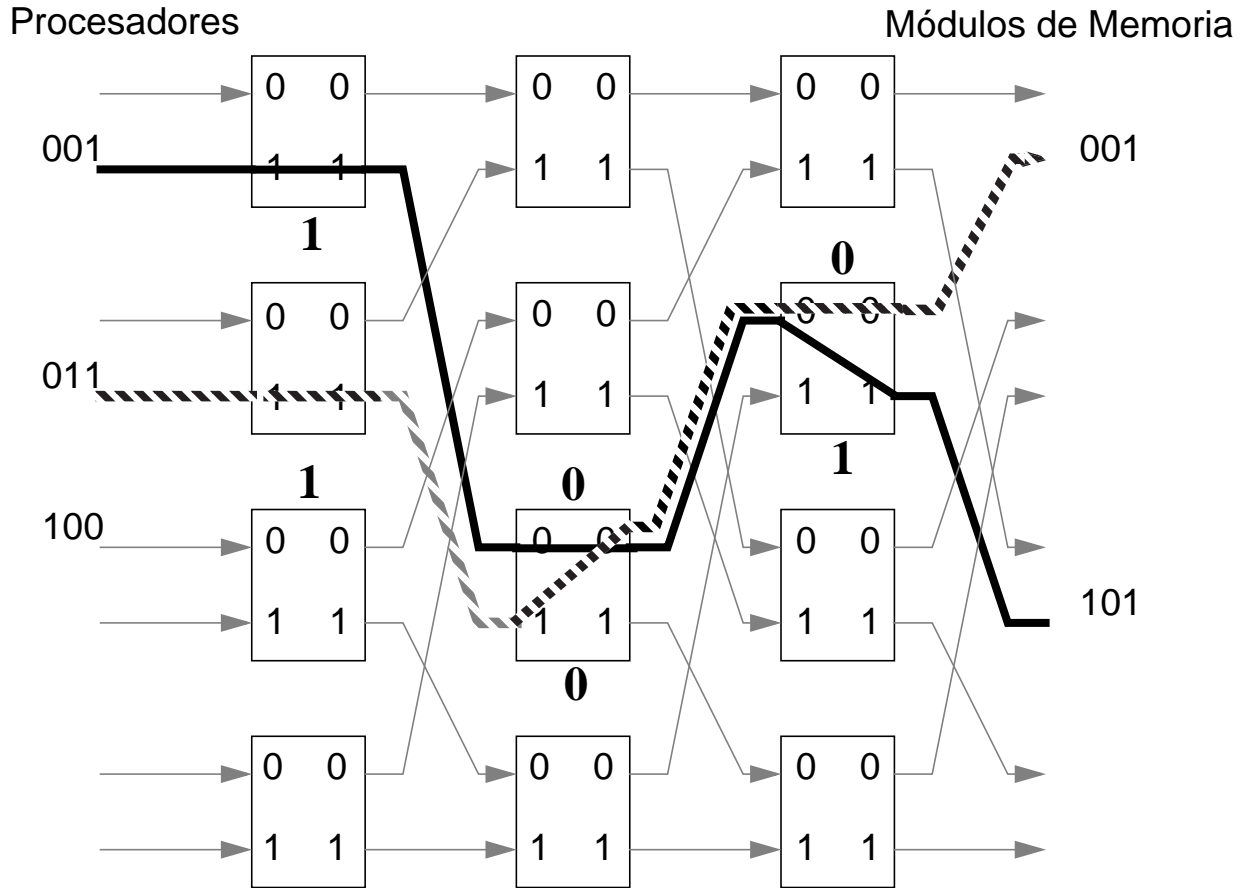
Redes de Interconexión

□ Ejemplo de Red Multietapa: La Red Omega



Redes de Interconexión

❑ Encaminamiento en la Red Omega



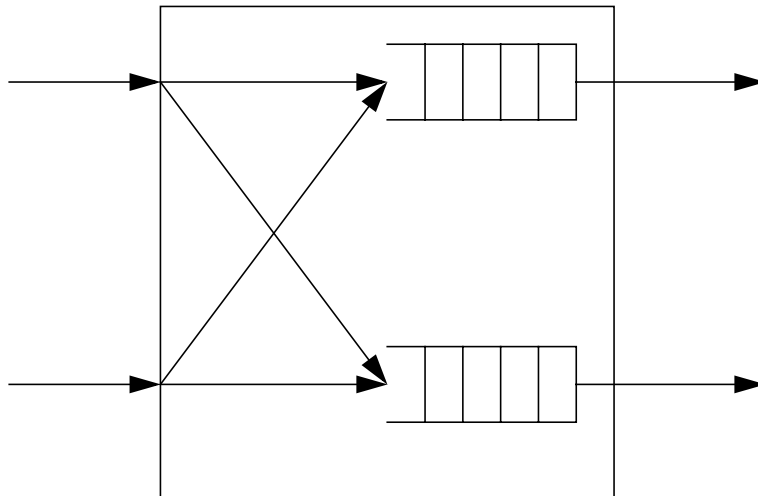
El encaminamiento se controla mediante los bits que identifican el módulo de memoria destino.

Se produce contención cuando dos mensajes requieren el mismo canal.

Redes de Interconexión

❑ Soluciones al Problema de la Contención en Redes Multietapa

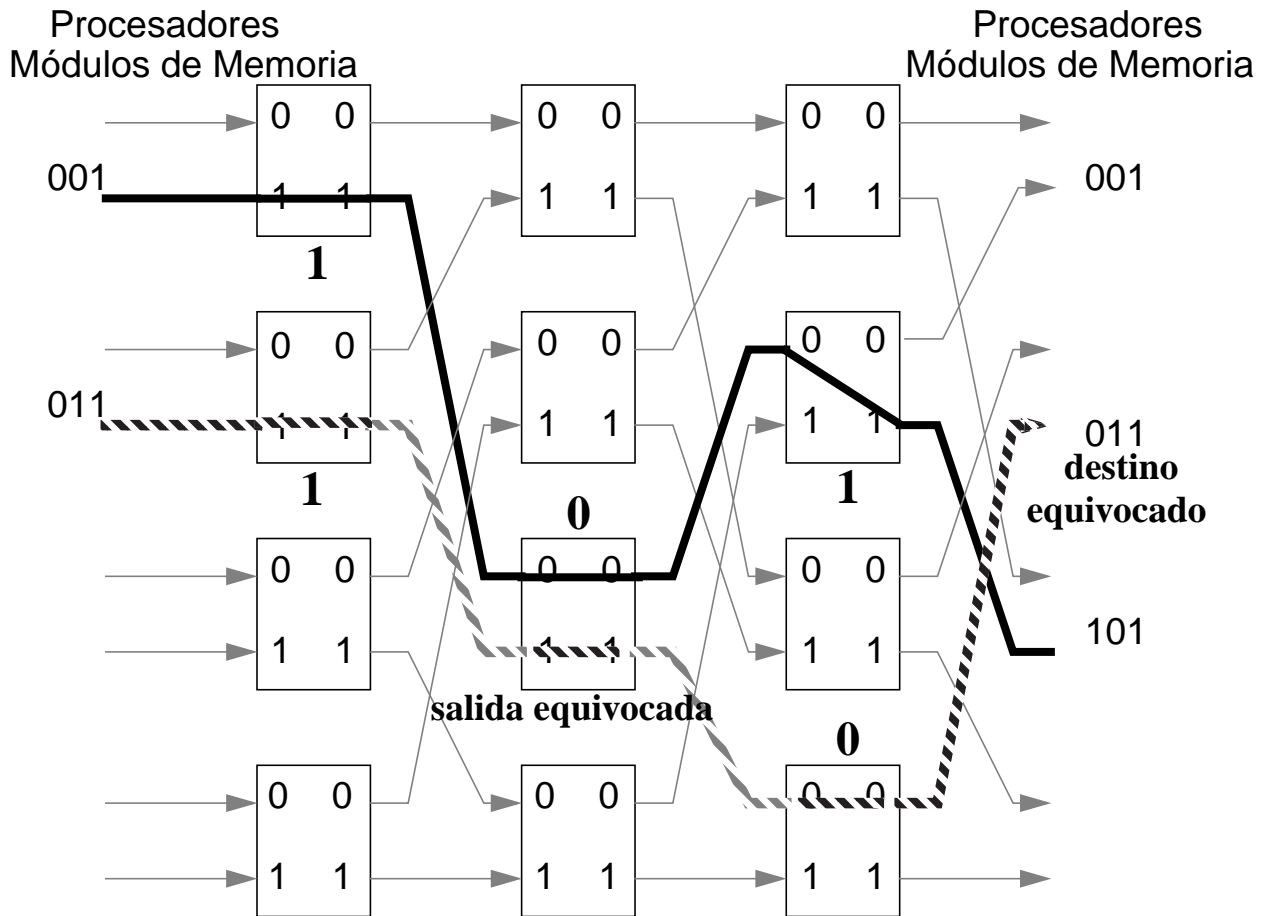
- ✓ Rechazar uno de los mensajes. Debe informarse al procesador que realizó el acceso rechazado para que lo reintente más tarde.
- ✓ Usar buffers en los elementos de conmutación.



Redes de Interconexión

❑ Soluciones al Problema de la Contención en Redes Multietapa

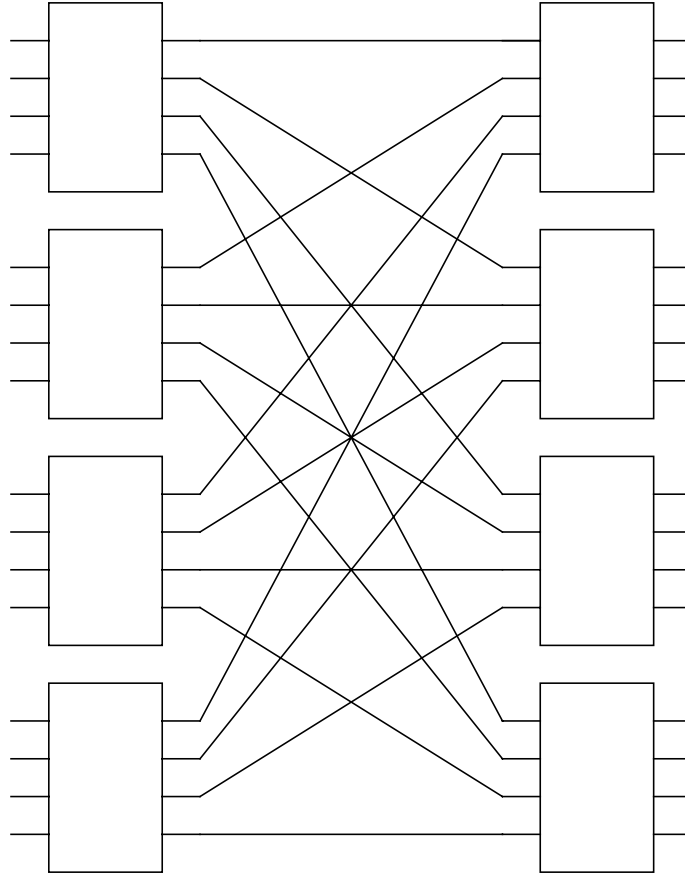
✓ Enviar el mensaje por el camino equivocado



El mensaje se envía de nuevo desde el destino equivocado.

Redes de Interconexión

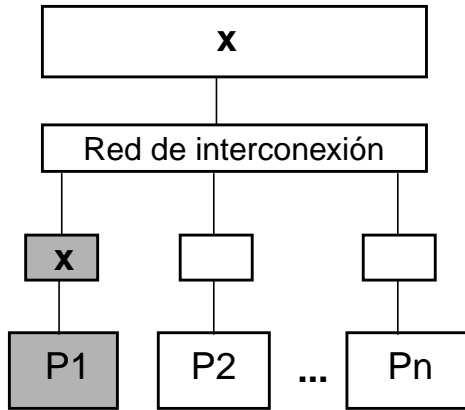
□ Ejemplo de Red Multietapa: La Red Butterfly



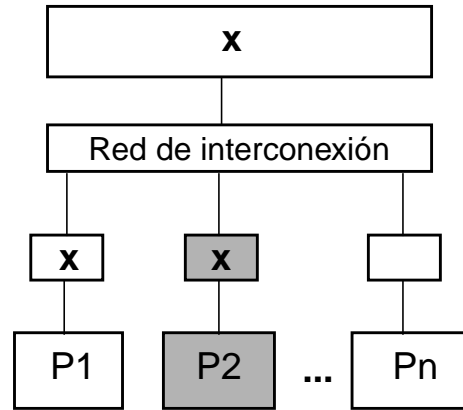
Los elementos de conmutación son más complejos pero la latencia se reduce.

Memorias Cache para SMP

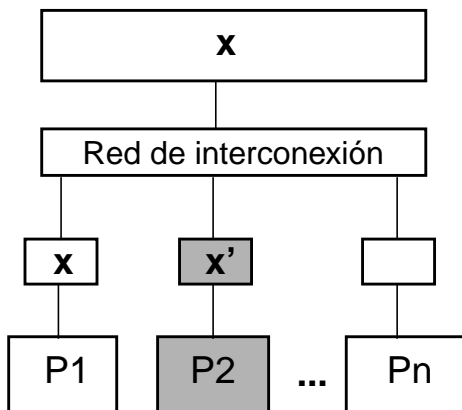
□ El Problema de la Coherencia de Memoria



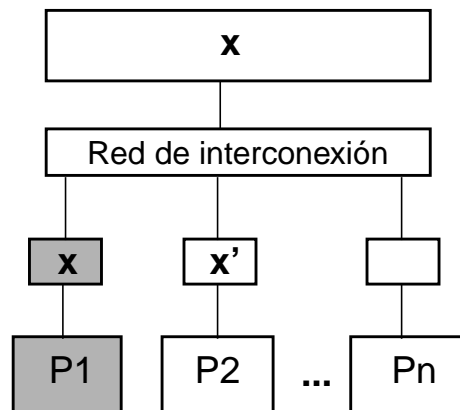
(a) P1 lee X



(b) P2 lee X



(c) P2 escribe X



(d) P1 lee el valor obsoleto de X

Debe incorporarse un mecanismo que permita que todos los procesadores tengan la misma visión de la memoria compartida.

Soluciones para el Problema de la Coherencia de Memoria

❑ Soluciones Hardware

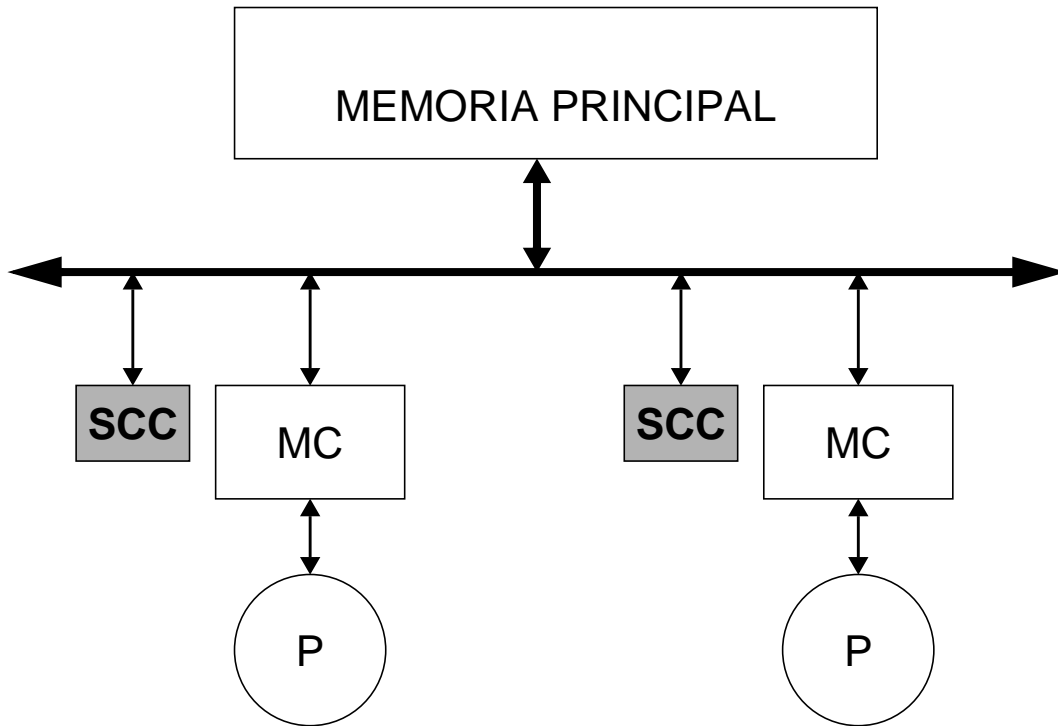
Un mecanismo hardware detecta y elimina las incoherencias.

❑ Soluciones Software

El programador es responsable de evitar las incoherencias. Para ello, se suministran primitivas software. Estas primitivas pueden requerir algún soporte hardware.

Soluciones Hardware

❑ Sistemas Basados en Controlador “Fisgoneador” del Bus (Snoopy Bus Controller)



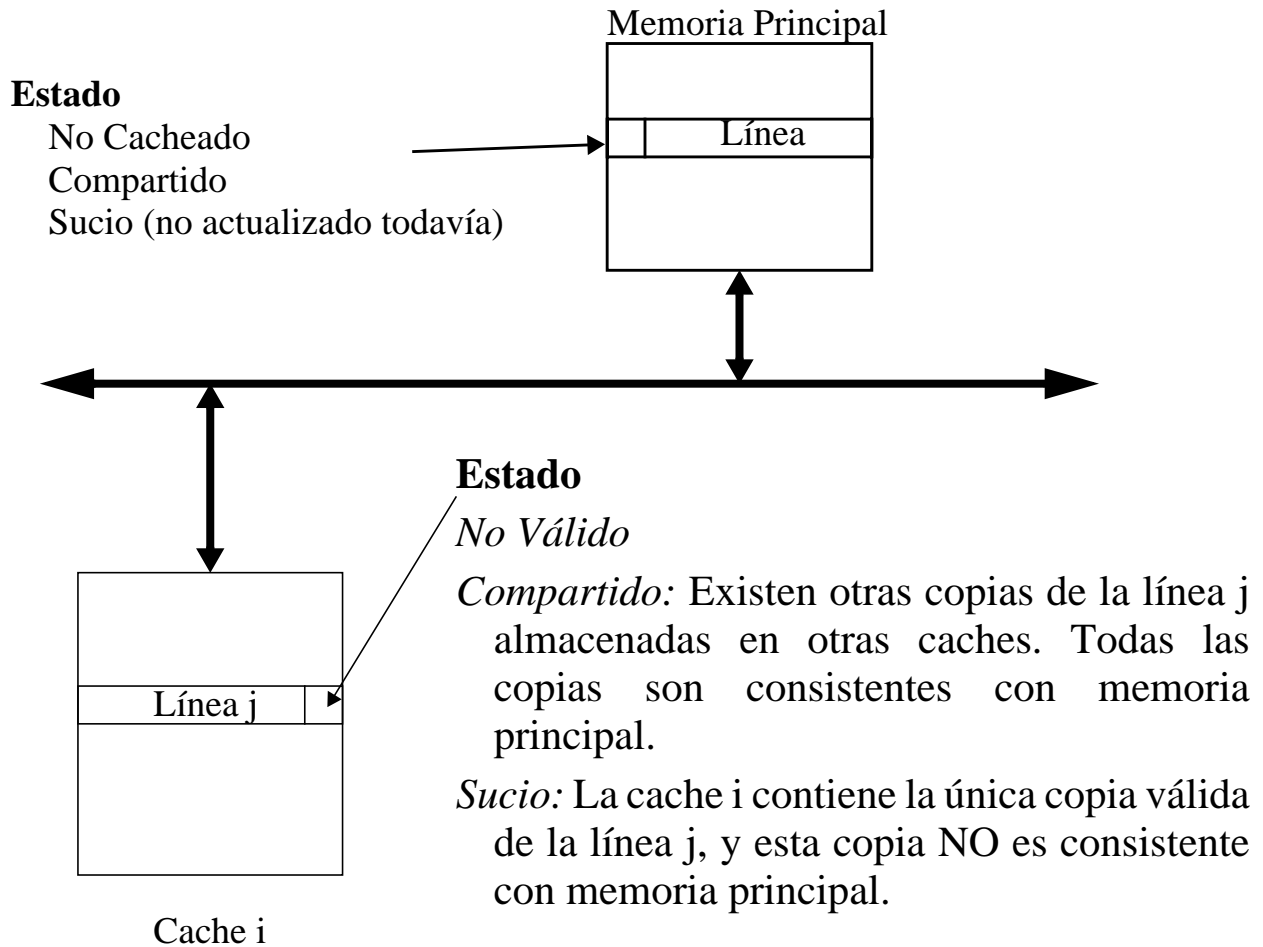
El circuito SCC observa todos los comandos enviados a través del bus (accesos a memoria y comandos de consistencia).

El protocolo implementado por los SCC (que se basa en intercambio de comandos de consistencia) garantiza la coherencia de memoria.

Soluciones Hardware

❑ Sistemas Basados en Controlador “Fisgoneador” del Bus

Un Protocolo Sencillo



Soluciones Hardware

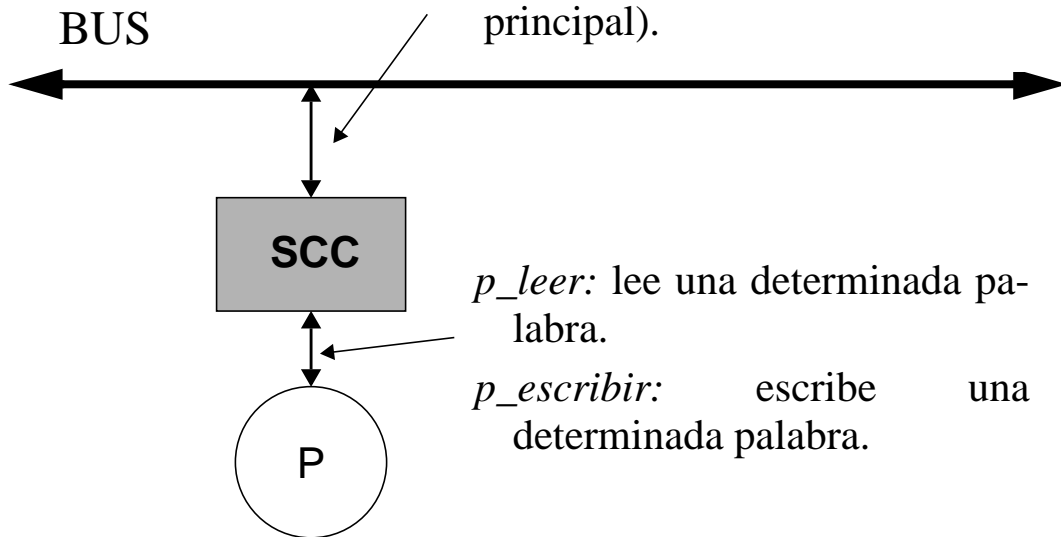
□ Sistemas Basados en Controlador “Fisgoneador” del Bus

Un Protocolo Sencillo

leer_línea: pide una copia de una determinada línea.

escribir_línea: escribe una determinada línea en memoria principal.

invalidar: invalida todas las copias de una determinada línea (incluida la copia de memoria principal).



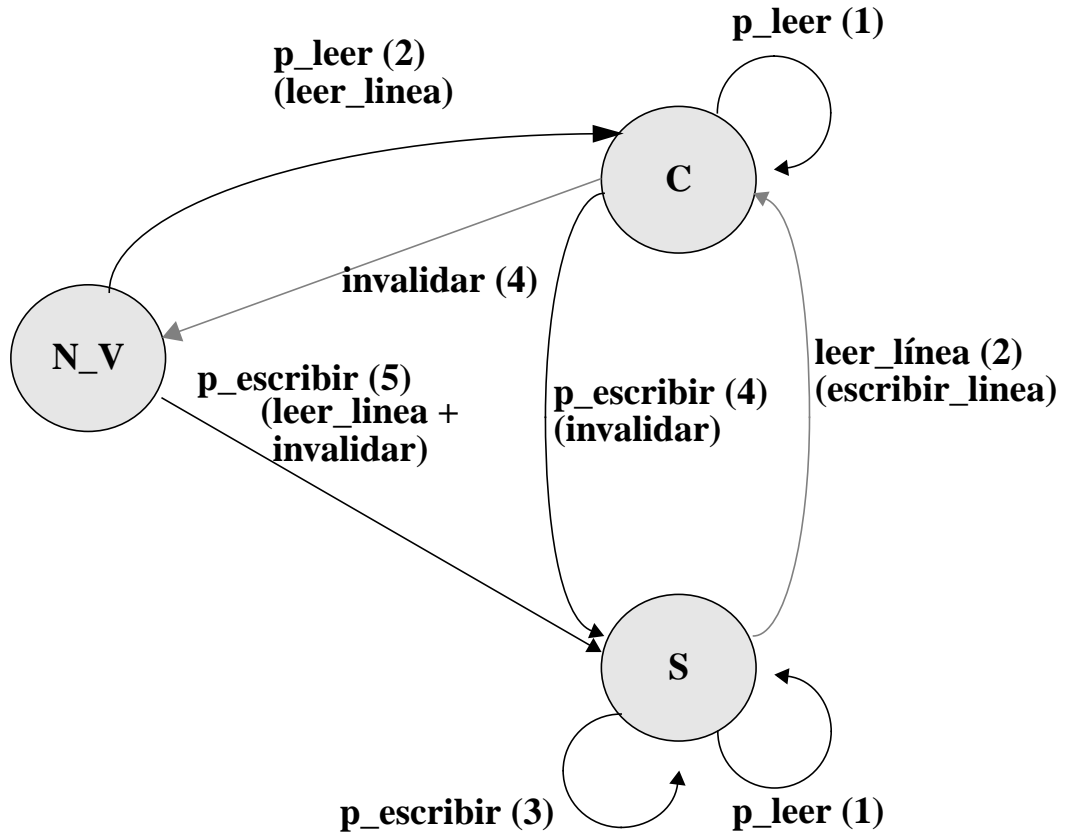
p_leer: lee una determinada palabra.

p_escribir: escribe una determinada palabra.

Soluciones Hardware

❑ Sistemas Basados en Controlador “Fisgoneador” del Bus

Un Protocolo Sencillo



Soluciones Hardware

❑ Sistemas Basados en Controlador “Fisgoneador” del Bus

Un Protocolo Sencillo

Acierto en Lectura: La petición es atendida por la cache local (1)

Fallo en Lectura: Envía **Leer_línea**. Si la línea no está cacheada o compartida, la memoria suministra la copia, que queda en estado **Compartido** (2). Si la línea está en estado Sucio entonces la única cache que tiene copia la suministra, se actualiza la memoria y la línea queda en estado **Compartido** (2).

Acierto en Escritura: Si el estado es **Sucio** entonces se realiza la escritura en la cache sin ninguna acción adicional de mantenimiento de coherencia (3). Si el estado es **Compartido** entonces envía **Invalidar**. El nuevo estado es **Sucio** (4).

Fallo en Escritura: Envía **Leer_línea**. Cuando llega la línea envía **Invalidar**. El nuevo estado es **Sucio** (5).



Soluciones Hardware

❑ Sistemas Basados en Controlador “Fisgoneador” del Bus

Tipos de Protocolos

Escritura e invalidación

Cuando se escribe sobre una línea de cache se invalidan todas las copias

- ✓ Write Once
- ✓ Synapse
- ✓ Berkeley
- ✓ Illinois

Escritura y actualización

Cuando se escribe sobre una línea de cache se actualizan todas las copias

- ✓ Firefly
- ✓ Dragon

Comparación

Cuando hay baja contención (varias escrituras seguidas sobre una línea compartida por parte de un solo procesador) es mejor *Escritura e invalidación* (menos tráfico en el bus). En casos de alta contención es mejor *Escritura y actualización* (menos fallos).



□ Un Modelo Sencillo

Instrucciones especiales

Invalidar: Invalida el contenido de la cache local

Ejemplo

Doall i=1,n

Invalidar

$Y(i) = W(i)*Y(i)$

enddo

Doall j=1,n

Invalidar

$X(j) = W(j)*Y(j)$

enddo

Invalidar

Do i=1,n

$Y(i) = Y(i-1)*Y(i)$

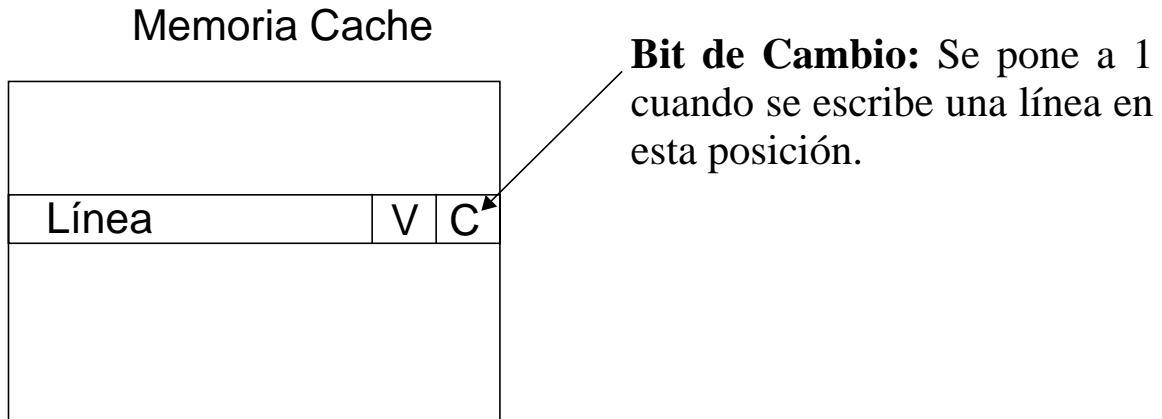
enddo



Soluciones Software

□ Una Solución Más Eficiente: Invalidación Selectiva

Soporte Hardware



Instrucciones especiales

Invalidar: Pone a 0 todos los bits de cambio de la cache local.

Leer_cache: El acceso es encaminado a través de la cache.

Leer_memoria: Si el bit de cambio es 1 entonces el acceso es encaminado a través de la cache. Si el bit de cambio es 0 entonces el acceso es servido por la memoria principal, ignorandose el contenido de la cache.

□ Una Solución Más Eficiente: Invalidación Selectiva

Ejemplo

Doall i=1,n

Invalidar

$Y(i) = \text{Leer_cache}(W(i)) * \text{Leer_cache}(Y(i))$

enddo

Doall j=1,n

Invalidar

$X(j) = \text{Leer_cache}(W(j)) * \text{Leer_memoria}(Y(j))$

enddo

Invalidar

Do i=1,n

$Y(i) = \text{Leer_memoria}(Y(i-1)) * \text{Leer_memoria}(Y(i))$

enddo



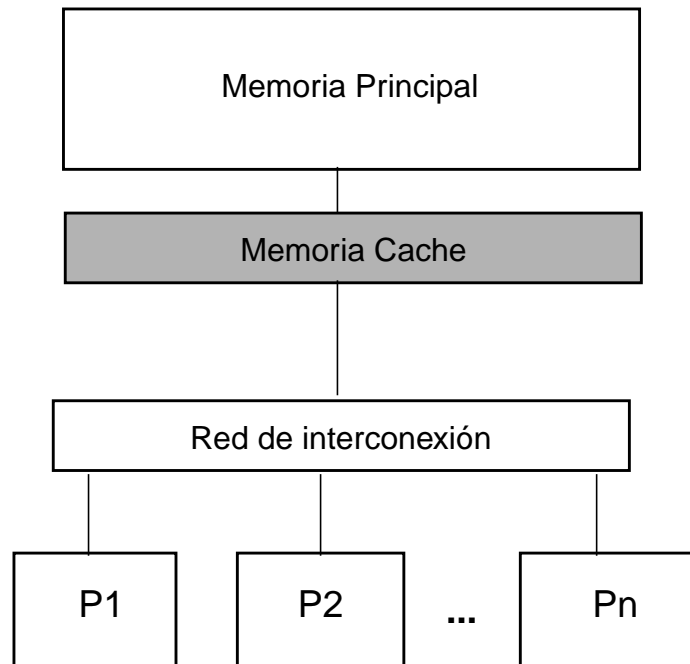
CEPBA



D A C

Memoria Cache Compartida

□ Organización



Memoria Cache Compartida

❑ **Ventajas**

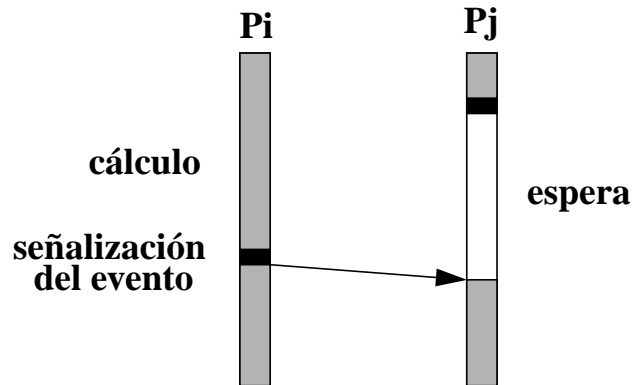
- ✓ Desaparece el problema de la coherencia
- ✓ La información que trae a la cache un procesador está disponible también para los otros
- ✓ Se comparten working sets: el espacio de memoria puede ser menor que la suma de los espacios de las caches privadas de la organización anterior
- ✓ Atractiva al considerar la implementación de un SMP en un chip

❑ **Problemas**

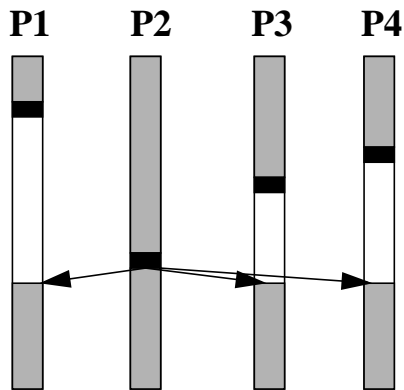
- ✓ Debe ser una memoria con muy alto ancho de banda
- ✓ Latencia mayor (hay que pasar siempre por la red de interconexión y la memoria es más grande)

Sincronización

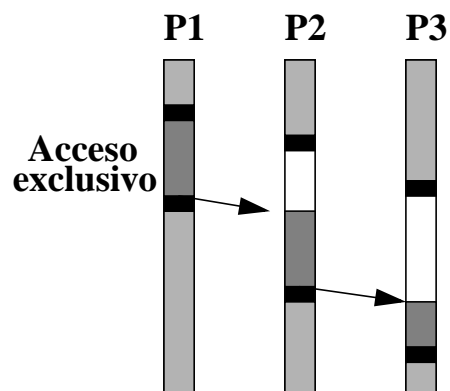
❑ Evento punto a punto



❑ Evento global

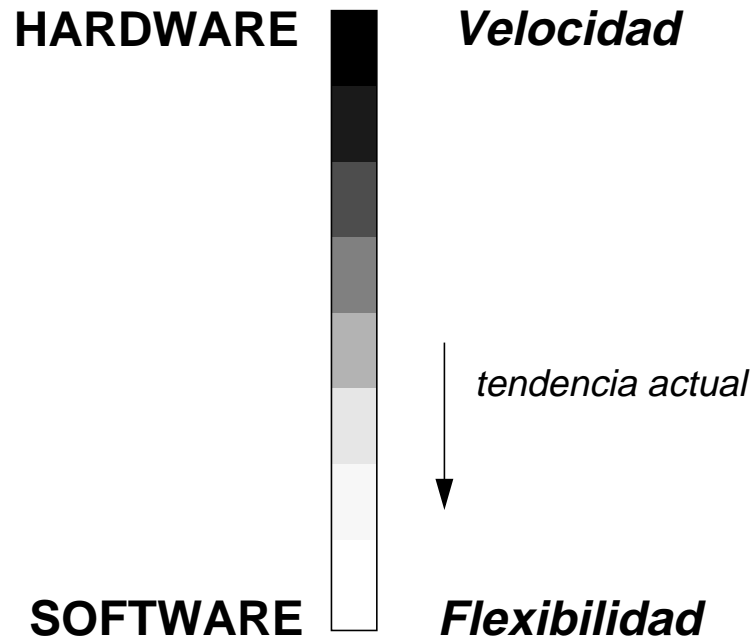


❑ Acceso exclusivo



Sincronización

□ Implementación de las primitivas



Acceso Exclusivo

❑ Instrucciones LOAD, STORE convencionales

```
lock:  ld  registro, flag
      cmp registro, #0
      bne lock
      st  flag, #1
```

acceso exclusivo

```
unlock: st  flag, #0
```

VERSIÓN INCORRECTA

variables compartidas {cont = 0, turno = 0, procA = fuera,
procB = fuera}

procesador A

```
procA:= dentro
if procB = dentro then
  if turno = 1 then
    procA := fuera
    repeat until turno = 0
    procA := dentro
  endif
  repeat until procB = fuera
endif
```

acceso exclusivo

```
turno := 1
procA:= fuera
```

procesador B

```
procB:= dentro
if procA = dentro then
  if turno =0 then
    procB := fuera
    repeat until turno = 1
    procB := dentro
  endif
  repeat until procA = fuera
endif
```

acceso exclusivo

```
turno := 0
procB:= fuera
```

VERSIÓN CORRECTA



CEPBA



D A C

Acceso Exclusivo

❑ Instrucciones especiales para sincronización

TEST-AND-SET

lock: t&s registro, flag → Copia flag en registro, y pone flag a 1.
 cmp registro, #0 OPERACIÓN INDIVISIBLE
 bne lock

acceso exclusivo

unlock: st flag, #0

LOAD-LOCKED (LL) y STORE-CONDITIONAL (SC)

lock: ll reg1, flag → Copia flag en reg1
 sc flag, reg2
 bz lock
 cmp reg1, #0 → Escribe reg2 en flag si ningun otro
 bne lock procesador ha escrito en flag después de
 ll. En caso contrario, no escribe y pone
 el flag z a 1.

acceso exclusivo

unlock: st flag, #0

❑ Problema

✓ Generan un alto tráfico en el bus

Acceso Exclusivo

❑ Mejoras

Test&Set con Backoff

```
lock:  t&s registro, flag
      cmp registro, #0
      be seguir
      ESPERA
      jmp lock
```

seguir:

acceso exclusivo

```
unlock: st  flag, #0
```

Test-and-Test&Set

```
lock:  ld  registro, flag
      cmp registro, #0
      bne lock
      t&s registro, flag
      cmp registro, #0
      bne lock
```

acceso exclusivo

```
unlock: st  flag, #0
```



Acceso Exclusivo

❑ Mejoras (cont.)

Ticket Lock

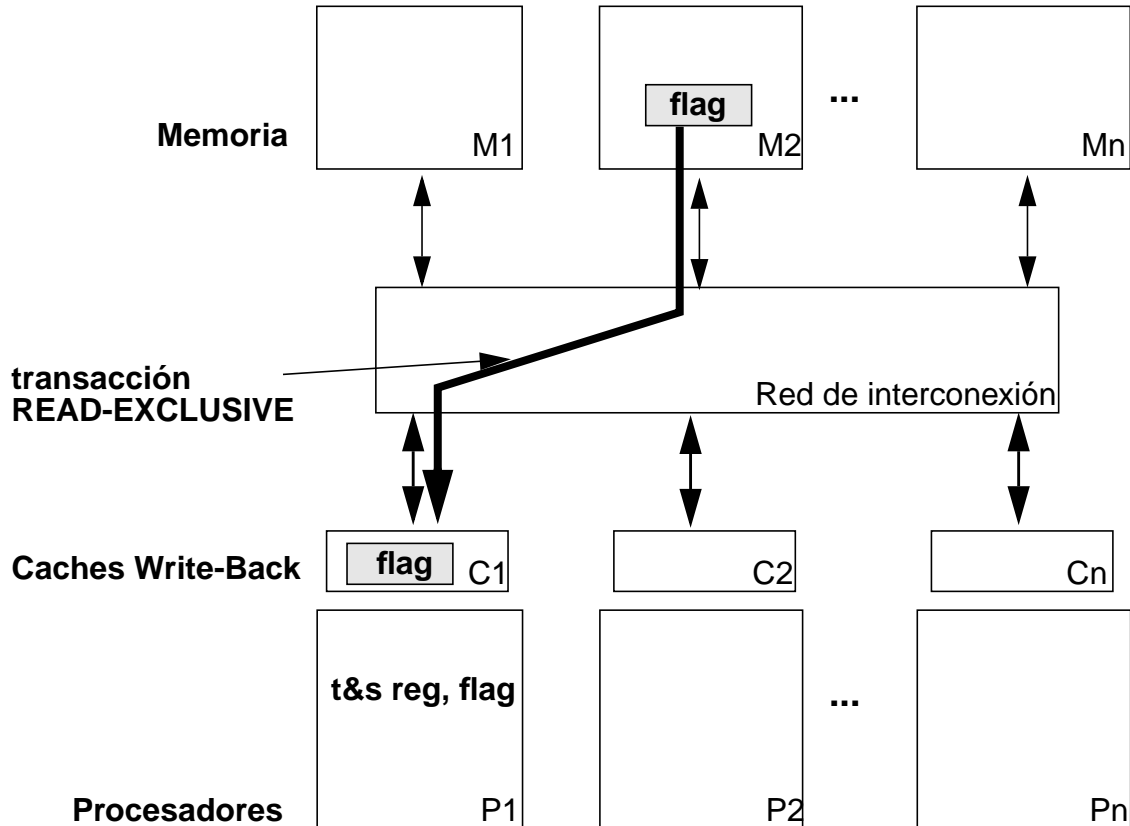
```
lock:   f&i registro, ticket → Copia ticket en registro, y suma 1 a  
        cmp registro, siguiente ticket.  
        acceso exclusivo OPERACIÓN INDIVISIBLE  
unlock: inc siguiente
```

Array-based Lock

```
lock:   f&i registro, indice  
        cmp A[registro], #0  
        bne lock  
        acceso exclusivo  
unlock: inc registro  
        st A[registro], #0
```

Acceso Exclusivo

□ Implementación de la Operación Indivisible



- ✓ Cuando el procesador obtiene la propiedad de la línea de cache, no la deja hasta que termina la operación

Evento Punto a Punto

❑ Instrucciones LOAD, STORE convencionales

variable compartida {flag = 0}

Procesador A

.
x := f(a)
flag := 1
.

Procesador B

.
while flag = 0 **do** nada **enddo**
y := f(x)
.

❑ Semáforos

{s = 0}

Procesador A

.
x := f(a)
signal (s)
.

Procesador B

.
wait (s)
y := f(x)
.

Semántica

Wait (s)

if s = 0 **then**
 bloquea la tarea en
 una cola
else s = 0

Signal (s)

if cola no vacía **then**
 desbloquear el primero
 de la cola
else s = 1

✓ Alto overhead



Evento Global: BARRIER

❑ Implementación Software

```
struct bar_type {
    int counter;
    struct lock_type lock;
    int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
    LOCK (bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0;
    bar_name.counter++;
    UNLOCK (bar_name.lock);
    if (bar_name.counter == p) {
        bar_name.counter = 0;
        bar_name.flag = 1;
    }
    else
        while (bar_name.flag == 0) {}
}
```

VERSIÓN INCORRECTA

✓ Problemas cuando se usa un mismo barrier dos veces seguidas



Evento Global: BARRIER

❑ Implementación Software

```
struct bar_type {
    int counter;
    struct lock_type lock;
    int flag = 0;
} bar_name;

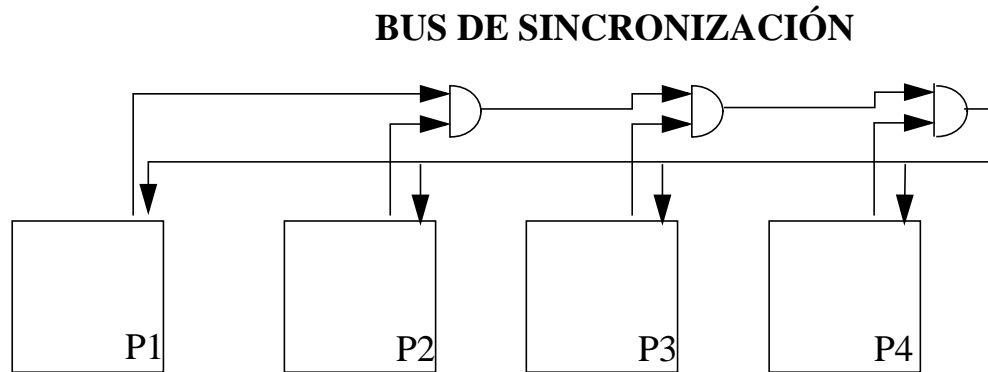
BARRIER (bar_name, p)
{
    local_sense = ! (local_sense);
    LOCK (bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0;
    bar_name.counter++;
    UNLOCK (bar_name.lock);
    if (bar_name.counter == p) {
        bar_name.counter = 0;
        bar_name.flag = local_sense;
    }
    else
        while (bar_name.flag != local_sense) {}
}
```

VERSIÓN CORRECTA



Evento Global: BARRIER

□ Implementación Hardware



✓ Mas rápido

✓ Menos flexible