

# Arquitectura de Computadores



## TEMA 6

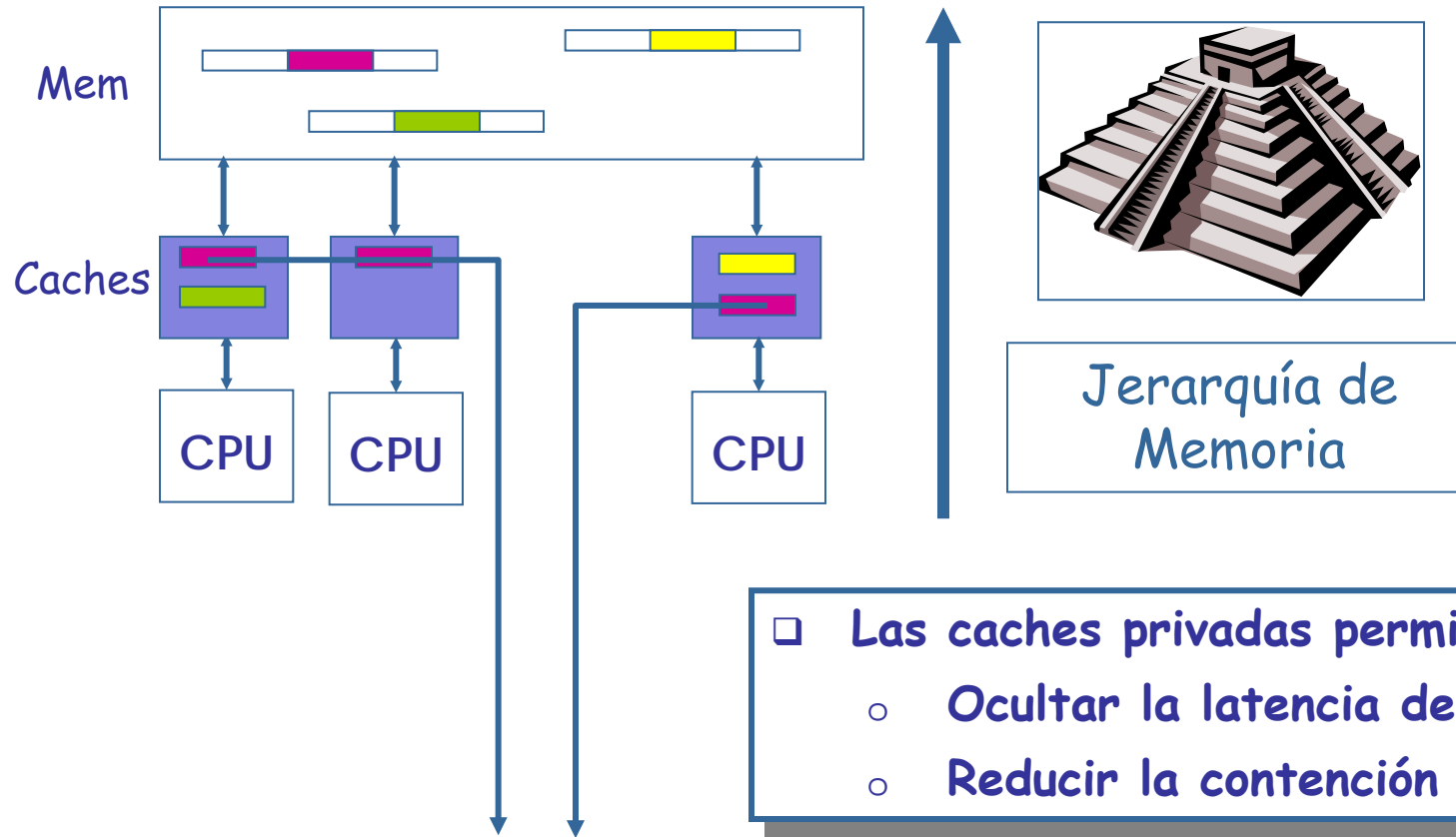
### Multiprocesadores: Coherencia, sincronización y consistencia

**D**EPARTAMENTO DE  
**A**RQUITECTURA DE **C**OMPUTADORES  
Y **A**UTOMÁTICA

Curso 2024-2025

- ❑ Jerarquía de memoria extendida
- ❑ El problema de la coherencia cache
- ❑ Protocolos de coherencia "Snoopy"
  - o Protocolo Snoopy de dos estados
  - o Protocolo MSI: invalidación de tres estados
  - o Protocolo MESI: invalidación de cuatro estados
- ❑ Coherencia cache basada en directorio
  - o Directorio plano basado en memoria
  - o Directorio plano basado en cache
- ❑ Sincronización
- ❑ Consistencia de memoria
- ❑ Bibliografía
  - o Cap 5 y 8 de Culler & Singh, "Parallel Computer Architecture: A Hardware/Software Approach", 1999
  - o Cap 5 de Hennessy & Patterson, 5th ed., 2012

# Jerarquía de memoria extendida



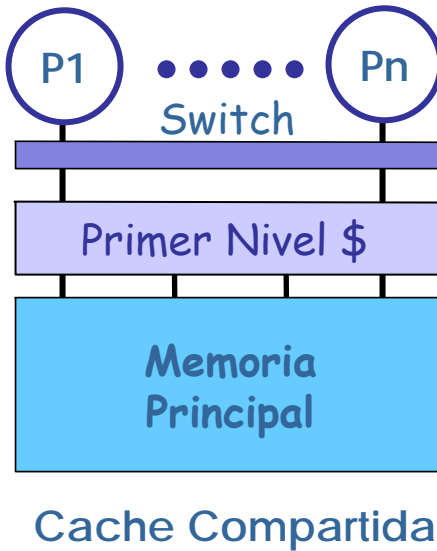
- ❑ Las caches privadas permiten:
  - Ocultar la latencia de los accesos
  - Reducir la contención

**Replicación:**  
**Problema de coherencia**

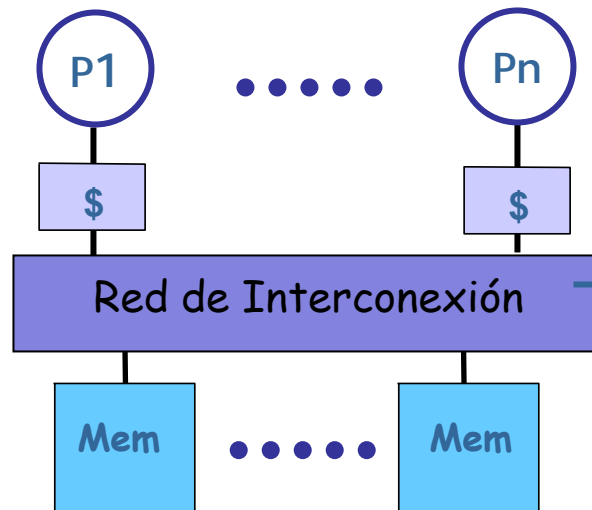
Variables Compartidas

# Alternativas de implementación

Escalabilidad

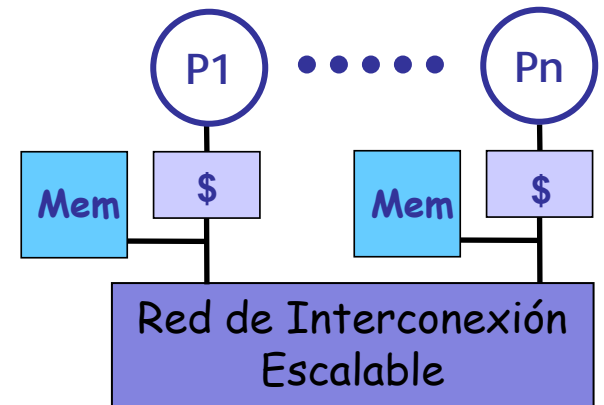


Lectura: Culler 5.1



- Bus Compartido
- Red punto a punto

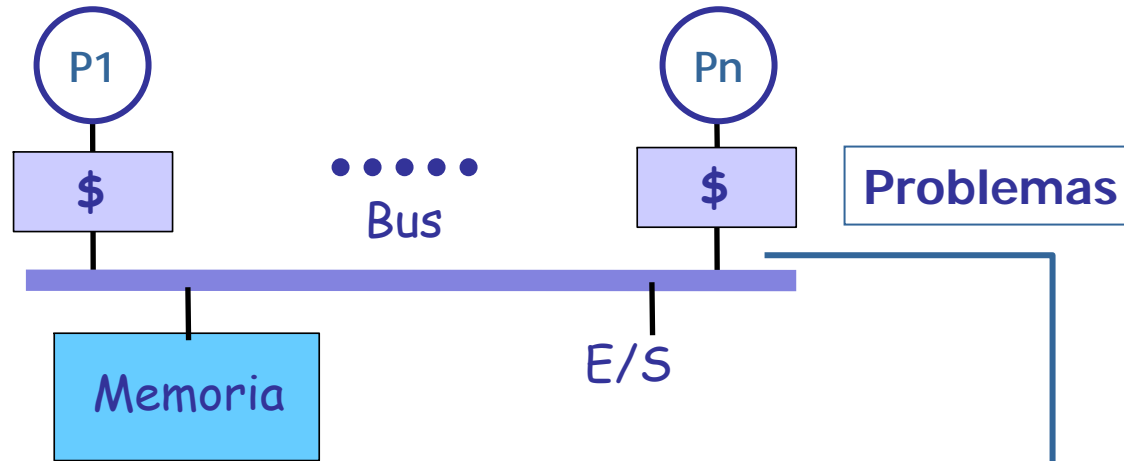
UMA, SMP



NUMA: Memoria Distribuida

# Alternativas de implementación

- ❑ Una alternativa ampliamente usada: SMP basado en Bus Compartido



- ❑ Servidores departamentales
- ❑ Estaciones de trabajo
- ❑ Bloques de construcción básicos sistemas de gran escala
- ❑ Soporte en microprocesadores de propósito general

# Modelo intuitivo de memoria

Lectura: Culler 5.2

## ❑ Programa secuencial

- o Las posiciones de memoria se utilizan para "comunicar valores" entre distintos puntos del programa. Cuando se lee una posición se devuelve el último valor escrito en ella

## ❑ Espacio de direcciones compartido en sistemas con un único procesador (Multiprogramación)

- o Cuando se lee una posición se devuelve el último valor escrito en ella, independientemente del proceso (thread) que realizó la última escritura sobre dicha posición
- o Las caches no interfieren con el uso de múltiples procesos (threads) en un procesador, ya que todos ellos ven la memoria a través de la misma jerarquía

## ❑ Multiprocesadores de Memoria Compartida

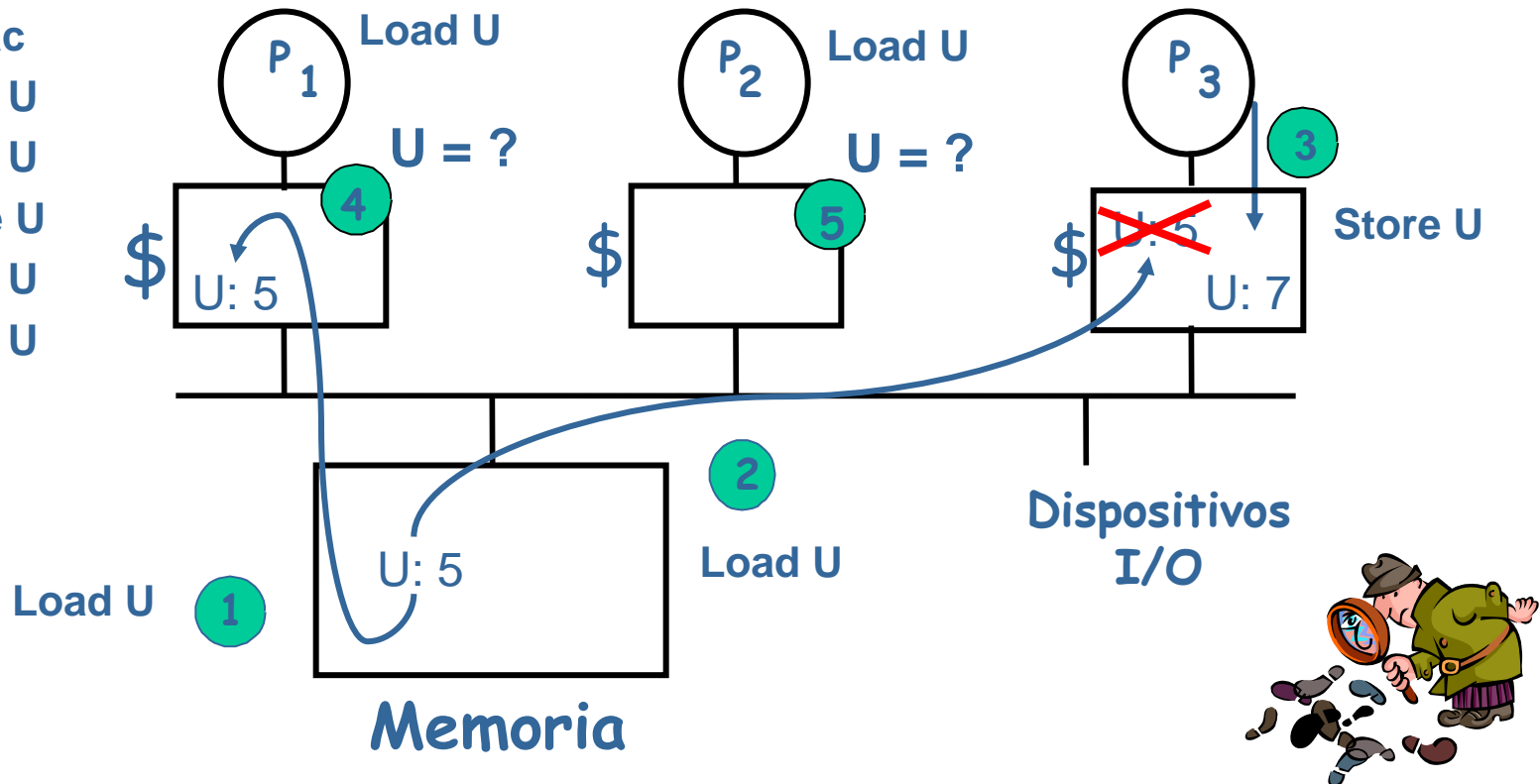
- o Objetivo: Nos gustaría que el **resultado de ejecutar un programa** que usa varios procesos (threads) **sea el mismo** independientemente de si los procesos se ejecutan en paralelo (multiprocesadores) o de forma entrelazada (uniprocador multiprogramado).
- o Problema: Cuando dos o más procesos acceden a la memoria a través de caches diferentes existe el peligro de que vean valores inconsistentes

# Problema de la coherencia cache

## Transacciones

(Proc) Operac

1. (P1) Load U
2. (P3) Load U
3. (P3) Store U
4. (P1) Load U
5. (P2) Load U



Protocolos de coherencia de cache



Siguen la pista del estado de cualquier bloque de datos compartido para evitar incoherencias

# Soluciones de “Grano Grueso”

- ❑ Problemas de coherencia cache en uniprosesadores:  
**operaciones de E/S a través de dispositivos DMA**
  - o Dispositivo DMA escribe en memoria: El procesador puede seguir viendo valores antiguos en la cache.
  - o Dispositivo DMA lee de memoria: El DMA puede leer un valor antiguo en el caso de que se utilice write-back
- ❑ Alternativas (dado que las operaciones de E/S son mucho menos frecuentes que las operaciones de acceso a memoria)
  - o Evitar usar la cache:
    - Los segmentos de memoria involucrados en operaciones de E/S se marcan como **No-Cacheables** o bien se utilizan operaciones load/store no-cacheables
  - o Sacar de la cache antes de E/S (**soporte del SO**):
    - Las páginas de memoria involucradas en cualquier operación de E/S son eliminadas de la cache (**flush**) previamente por el SO
  - o Usar la cache para E/S:
    - El tráfico de E/S pasa por todos los niveles de la jerarquía de memoria
    - Problema: el contenido de la cache puede “corromperse” con datos que no son de interés (al menos inmediato) para el procesador

## ❑ En multiprocesadores

- o La escritura o lectura de variables compartidas es un evento frecuente.
- o No es práctico:
  - Deshabilitar la cache para datos compartidos
  - Invocar al SO en cada referencia a una variable compartida

## ❑ Todos los $\mu$ Procesadores actuales proporcionan mecanismos para soportar la coherencia

- o La coherencia de caches se tiene considerar como una cuestión fundamental de diseño hw
- o Transparente al software

## ❑ ¿Qué significa que un sistema de memoria es coherente?

- o Una operación de lectura retorna siempre el **último** valor que fue escrito en la posición de memoria correspondiente, independientemente del procesador que efectúa la lectura o escritura
- o Dos aspectos clave:
  - 1) Qué valor debe ser devuelto en una lectura Coherencia.
  - 2) Cuándo un valor escrito debe ser devuelto por una operación de lectura Consistencia

## ❑ ¿Cuándo un sistema de memoria es coherente?

Lectura: H&P 5th ed, p. 352-353

- o Una lectura del procesador P a la posición X de memoria, que sigue a una escritura de P en X, sin ninguna otra escritura debe retornar el valor escrito por P.
- o Una lectura por un procesador a la posición X que sigue a una escritura de **otro procesador** en X, retorna el valor escrito si no hay ninguna otra escritura y están suficientemente separadas en el tiempo.
- o **Serialización** de escrituras. Todas las escrituras a la misma posición deben verse en el orden correcto.
  - Dos escrituras hechas por dos procesadores sobre la misma posición, X, son vistas por todos los procesadores en el mismo orden.

## ❑ Serialización de escrituras: ejemplo

Si **P1** ejecuta:

- o LD R1, **X**

- y se obtiene en R1 el valor V1 escrito por el procesador **P2** en la posición de memoria X

Y después **P1** ejecuta:

- o LD R2, **X**

- y se obtiene en R2 el valor V2 escrito por el procesador **P3** en la posición de memoria X

...entonces es imposible que cualquier otro procesador,  $P_i$ , que lea la posición de memoria X obtenga primero el valor V2 y después el valor V1.

# Políticas (o protocolos) para mantener la coherencia

## ❑ Invalidación en Escritura / Coherencia Dinámica

- o Al escribir en un bloque se invalidan todas las otras copias (Múltiples lectores , un solo escritor)
- o Escrituras consecutivas al mismo bloque (no necesariamente a la misma palabra) efectuadas desde el mismo procesador se realizan localmente (No hay copias). Sólo es necesario una transacción en el medio de comunicaciones en múltiples escrituras al mismo dato
- o ¿Qué ocurre en un fallo de lectura?
  - Con write-through: la memoria esta siempre actualizada
  - Con write-back: es necesario búsqueda (snoop) en caches remotas para encontrar el último valor.
    - La cache dueña del último valor lo proporciona al solicitante (y a la Mp)

## ❑ Actualización en Escritura

- o Al escribir en un bloque se actualizan todas las copias
- o Típicamente con write-through, pocos procesadores
- o Escrituras consecutivas a la misma palabra requiere múltiples actualizaciones
- o ¿Qué ocurre en un fallo de lectura?
  - Se busca en la memoria. Siempre está actualizada

# Políticas (o protocolos) para mantener la coherencia

- ❑ Los protocolos usados para la invalidación o actualización dependen de la red de interconexión utilizada

Lectura: H&P 5th ed, p. 354-355

- ❑ Si la red de interconexión **permite broadcast** eficiente las operaciones de invalidación o actualización se pueden enviar de forma simultánea a todos los controladores.

- o **Protocolo Snoopy (buses): Observación del bus**

- Cada controlador de cache está observando los eventos que suceden en el bus y toma decisiones en consecuencia.
- Un controlador de cache envía eventos al bus en función de las peticiones de acceso a memoria que recibe de su procesador → Envía órdenes de **invalidación** o **actualización** de forma simultánea a todos los demás controladores.

- ❑ Si la red de interconexión **no permite broadcast** (o el broadcast no es eficiente), la invalidación o la actualización se envía únicamente a aquellas caches que tienen una copia del bloque.

- o **Protocolos Basados en Directorio:** Se utiliza un directorio (centralizado o distribuido) con una entrada por cada bloque en la que se indica en qué caches existe copia y en qué estado.

# Protocolos Snoopy

- ❑ El Arbitraje del bus Impone un Orden.
  - o Las Transacciones del Bus son visibles en el mismo orden por todos los controladores de cache.
- ❑ Observar/Espiar el bus

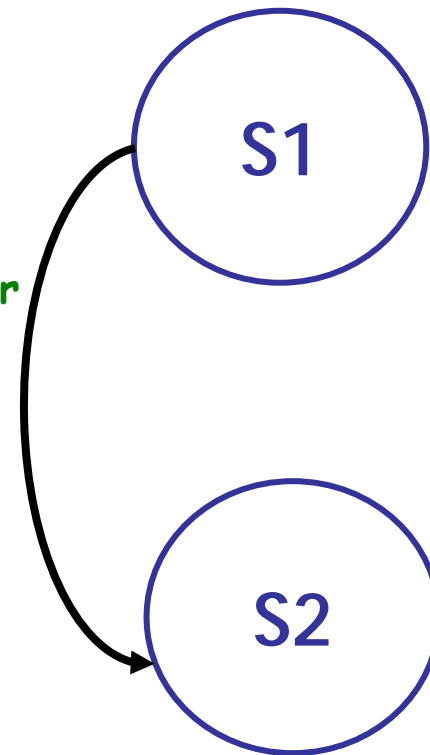


Sólo es necesario extender la funcionalidad del controlador de cache

- El estado de cada bloque mapeado en la cache sigue estando en el directorio
- Los cambios de estado son provocados por:
  - a) operaciones Ld/St del procesador
  - b) **Transacciones relevantes observadas en el bus**

- ❑ Diagrama de estados de un bloque de cache: convenio de representación
  - o Cada transición es disparada por la observación de un evento en el procesador o el bus
  - o Una transición puede implicar que el controlador ejecute alguna acción sobre su cache, u ordene una cierta acción al bus

Notación:  
Op del Procesador/ Acción a realizar  
o bien  
Evento en BUS / Acción a realizar



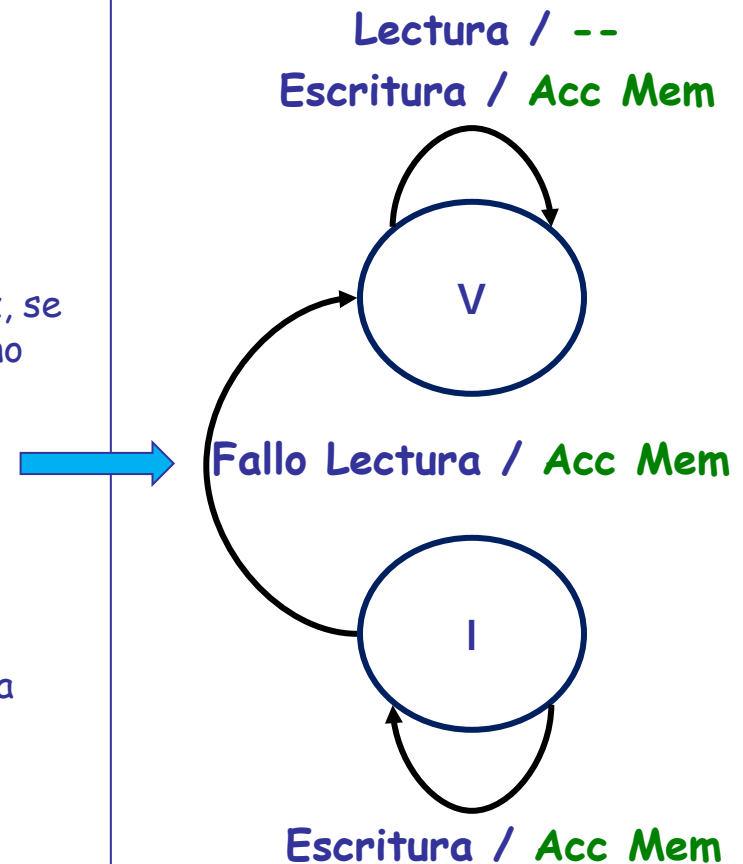
# Protocolo Snoopy de 2 Estados

- ❑ Aplicado a: Política de invalidación en escritura + Escritura directa (*write through*).
  - o Sin Asignación en Escritura (*no write allocate*)

## Ejemplo previo:

### Diagrama de Estados en un **Uniprocador**

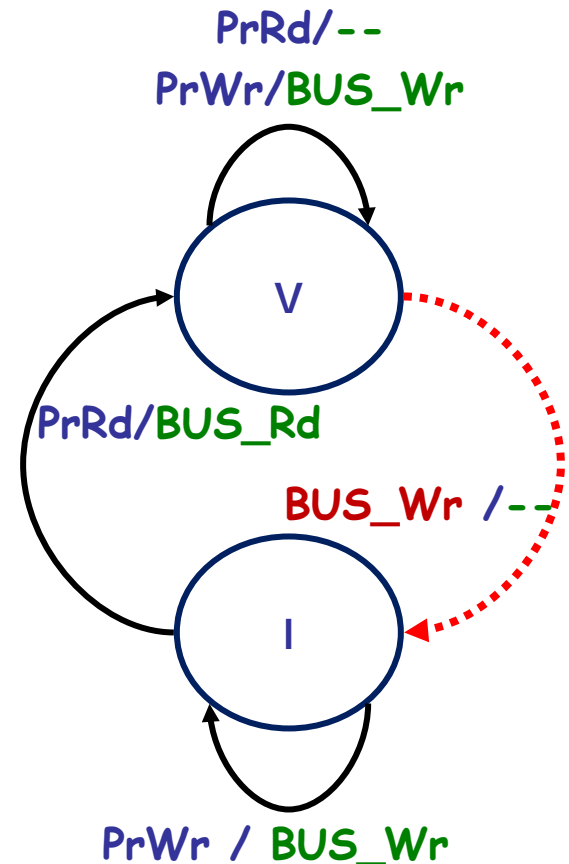
- Cada bloque tiene un estado (I,V)
  - Nota: Aunque solamente los bloques residentes en \$ tienen un bit de validez, se considera que el estado de los bloques no residentes es Inválido
- Estado inicial : bloques inválidos
- Fallo Lectura : Inválido → Válido
  - Puede generar reemplazamiento
- Escritura: no cambia el estado
  - En estado I (fallo), solo se escribe en la Memoria (*no write allocate*)
  - En estado V (acierto), se escribe en la Memoria y la \$ (*write through*)



# Protocolo Snoopy de 2 Estados

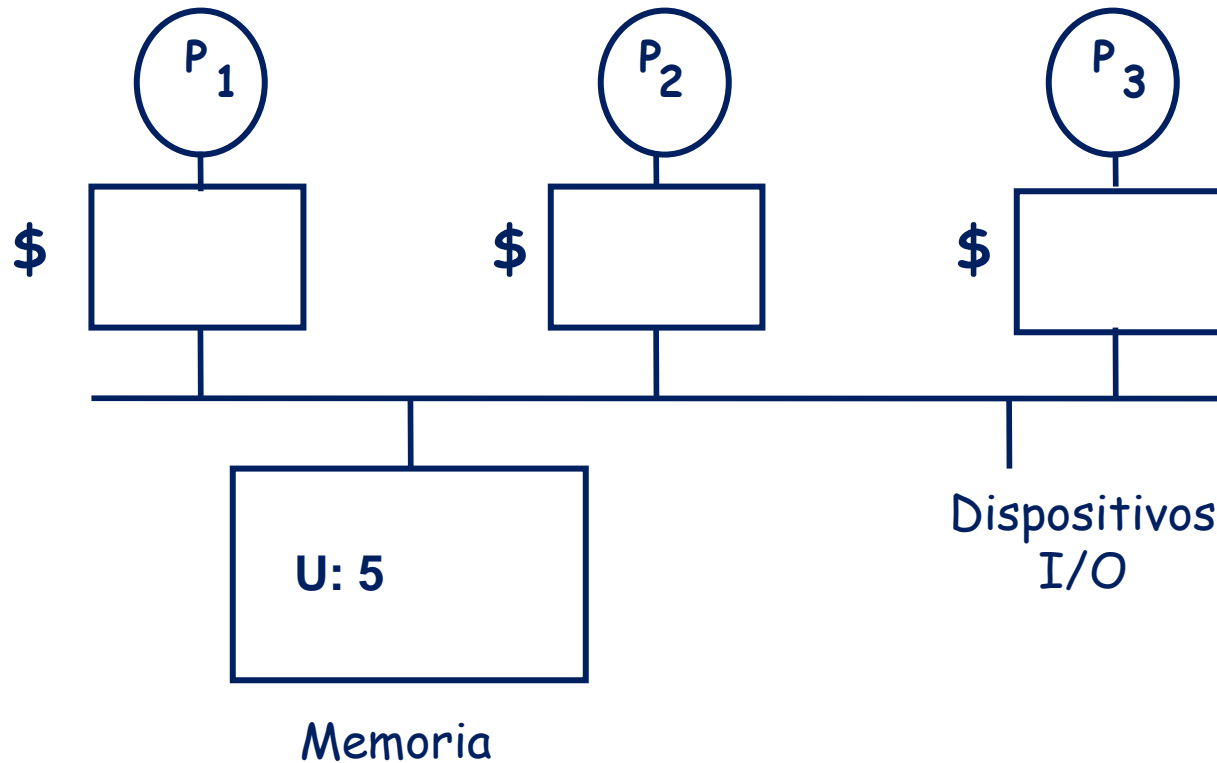
## Diagrama de Estados en Protocolo Snoopy

- Operaciones del Procesador
  - Lecturas: PrRd
  - Escrituras: PrWr
- Transacciones del BUS ordenadas por el controlador de \$
  - BUS\_Rd (fallo de lectura)
  - BUS\_Wr (relevante: actualiz. Mp)
- Transacciones en el BUS detectadas por el controlador de \$ (línea - - - - - )
  - BUS\_Wr (Si otro procesador ha escrito un bloque que está en esta \$: **Invalidar** bloque local)



# Protocolo *Snoopy* de 2 Estados

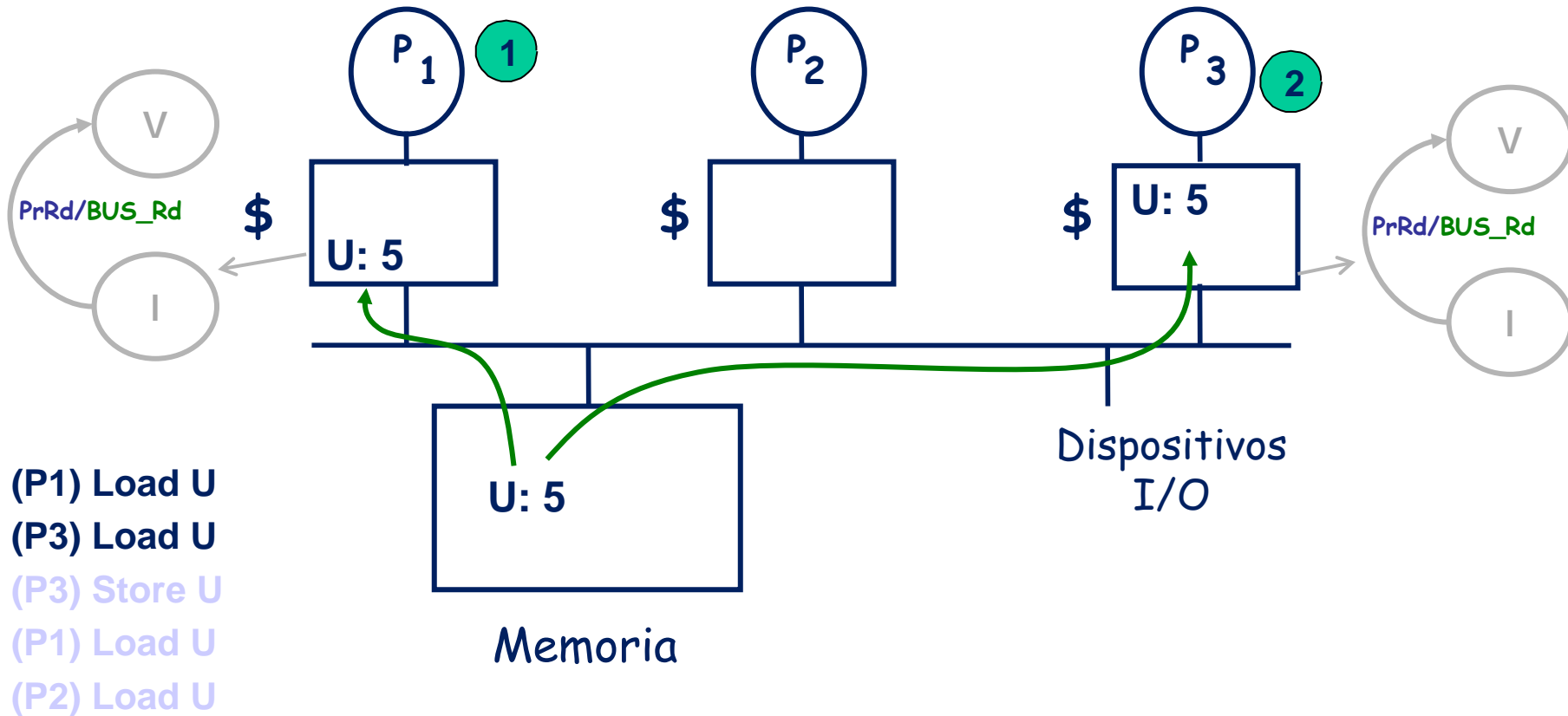
## □ Ejemplo: Escritura directa + Invalidación



(P1) Load U  
(P3) Load U  
(P3) Store U  
(P1) Load U  
(P2) Load U

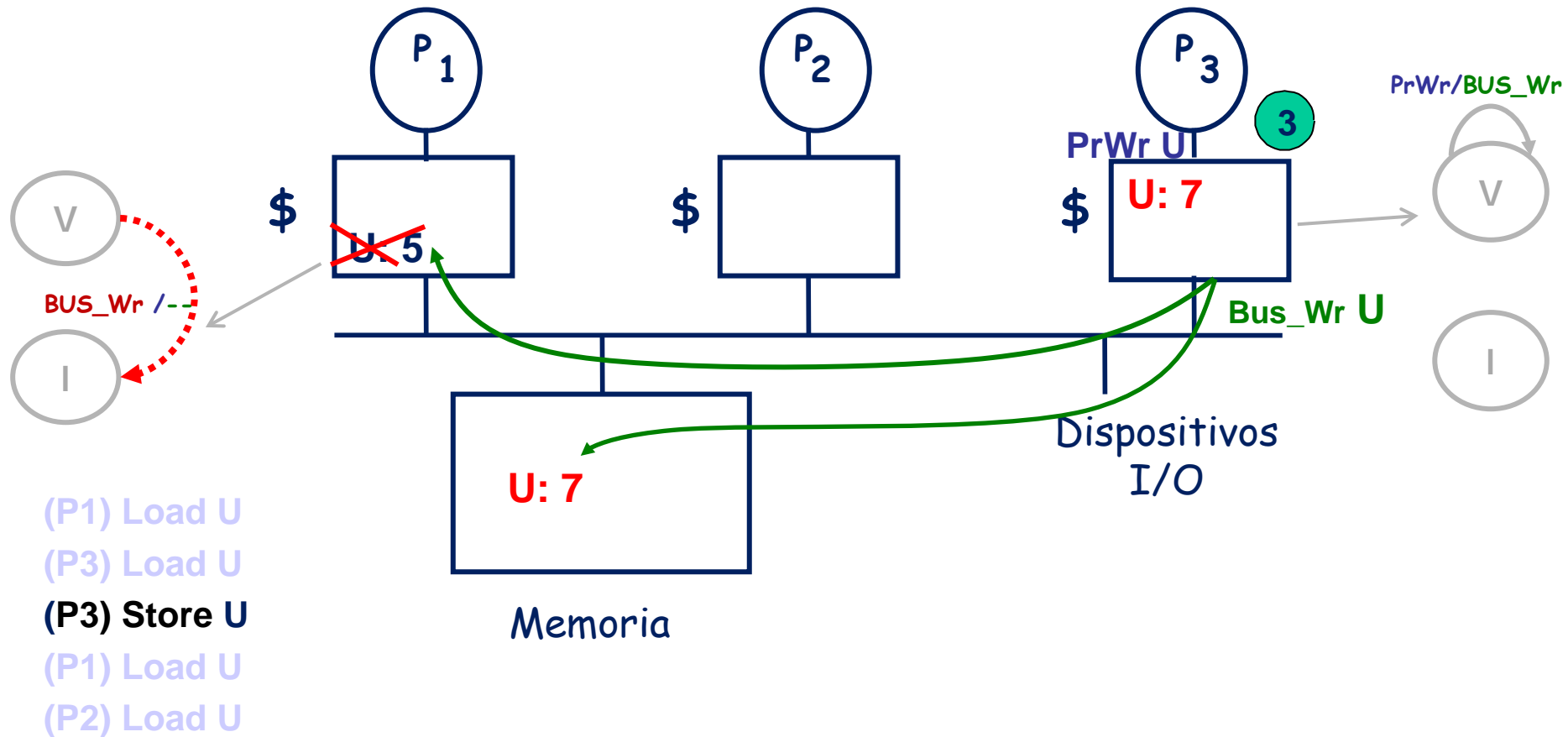
# Protocolo *Snoopy* de 2 Estados

## □ Ejemplo: Escritura directa + Invalidación



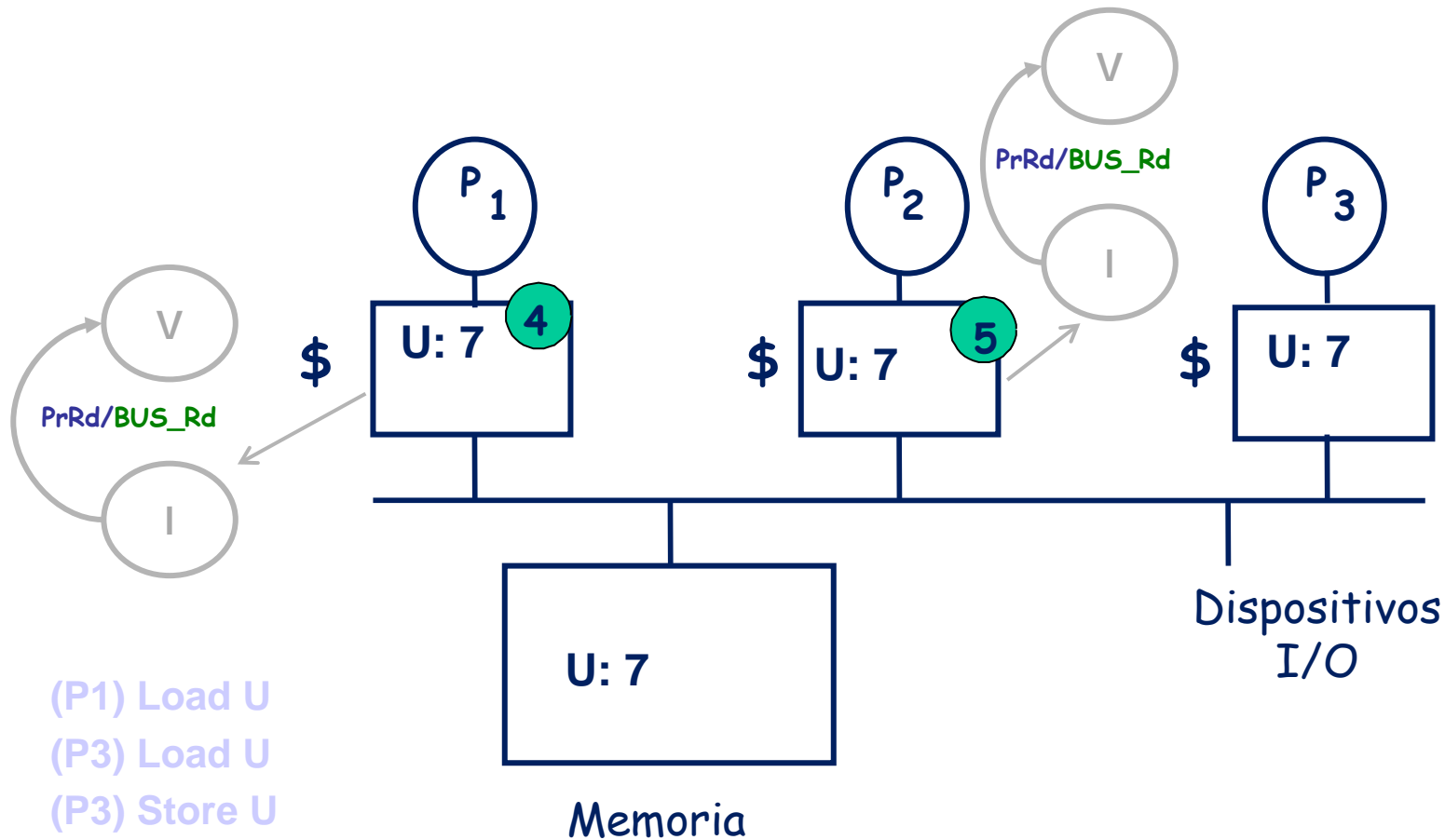
# Protocolo Snoopy de 2 Estados

## □ Ejemplo: Escritura directa + Invalidación



## Protocolo Snoopy de 2 Estados

## ❑ Ejemplo: Escritura directa + Invalidación



(P1) Load U  
(P3) Load U  
(P3) Store U  
**(P1) Load U**  
**(P2) Load U**

# Protocolo Snoopy de 2 Estados

- ❑ ¿Es coherente el Protocolo Snoopy de 2 estados?
  - o ¿Existe una Ordenación Secuencial de las operaciones de Memoria?

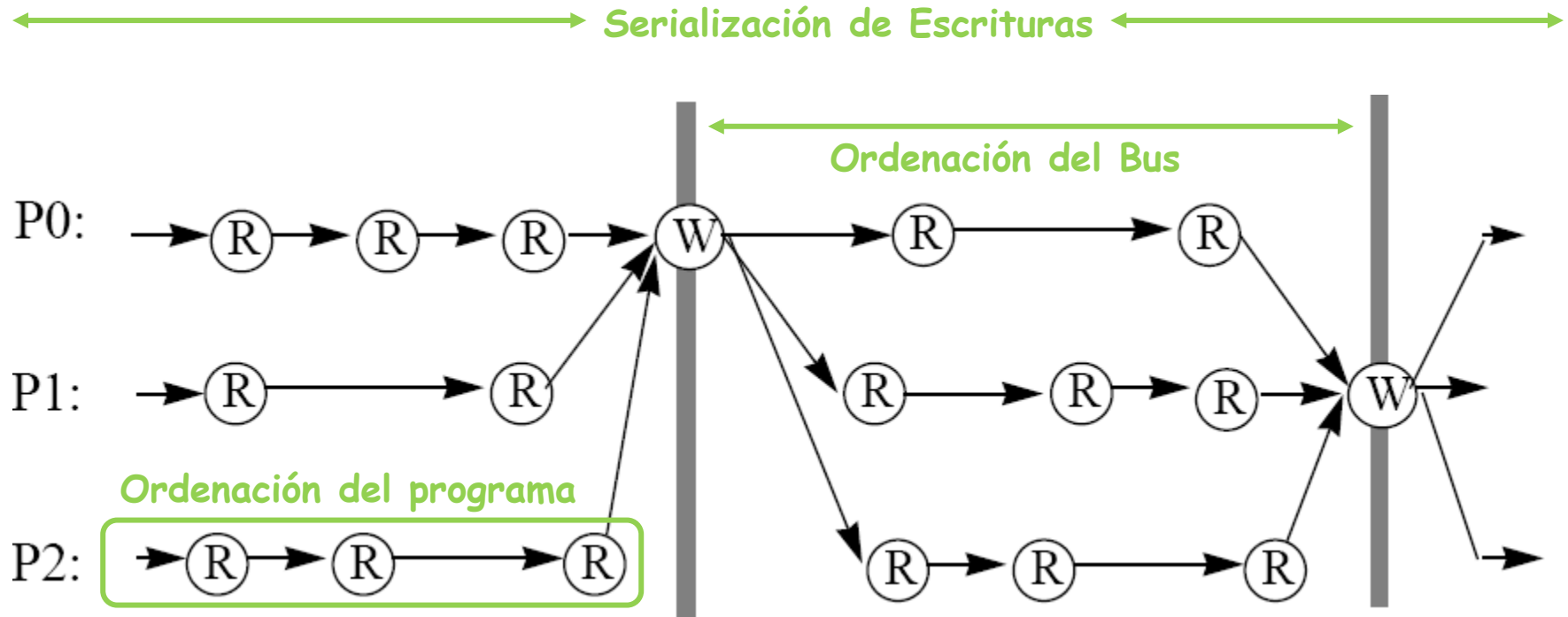
- o Escritura directa y 1 solo nivel de cache
- o Bus atómico
- o Operaciones atómicas

**Orden parcial de escrituras = Orden del bus**

- o Propagación de escrituras
  - Las escrituras son visibles a todos los controladores
- o Serialización de Escrituras
  - El arbitraje de bus define el orden con el que se efectúan las escrituras a todas las posiciones de memoria

# Protocolo Snoopy de 2 Estados

❑ ¿Es coherente el Protocolo Snoopy de 2 estados?



- ❑ Las transacciones del bus asociadas con escrituras segmentan los flujos individuales de cada programa
- ❑ Entre dos transacciones de escritura el orden de las lecturas en los distintos procesadores no está restringido
- ❑ Entre escrituras, cada programa hace sus lecturas de acuerdo con el orden del programa

# Protocolo *Snoopy* de 2 Estados

---

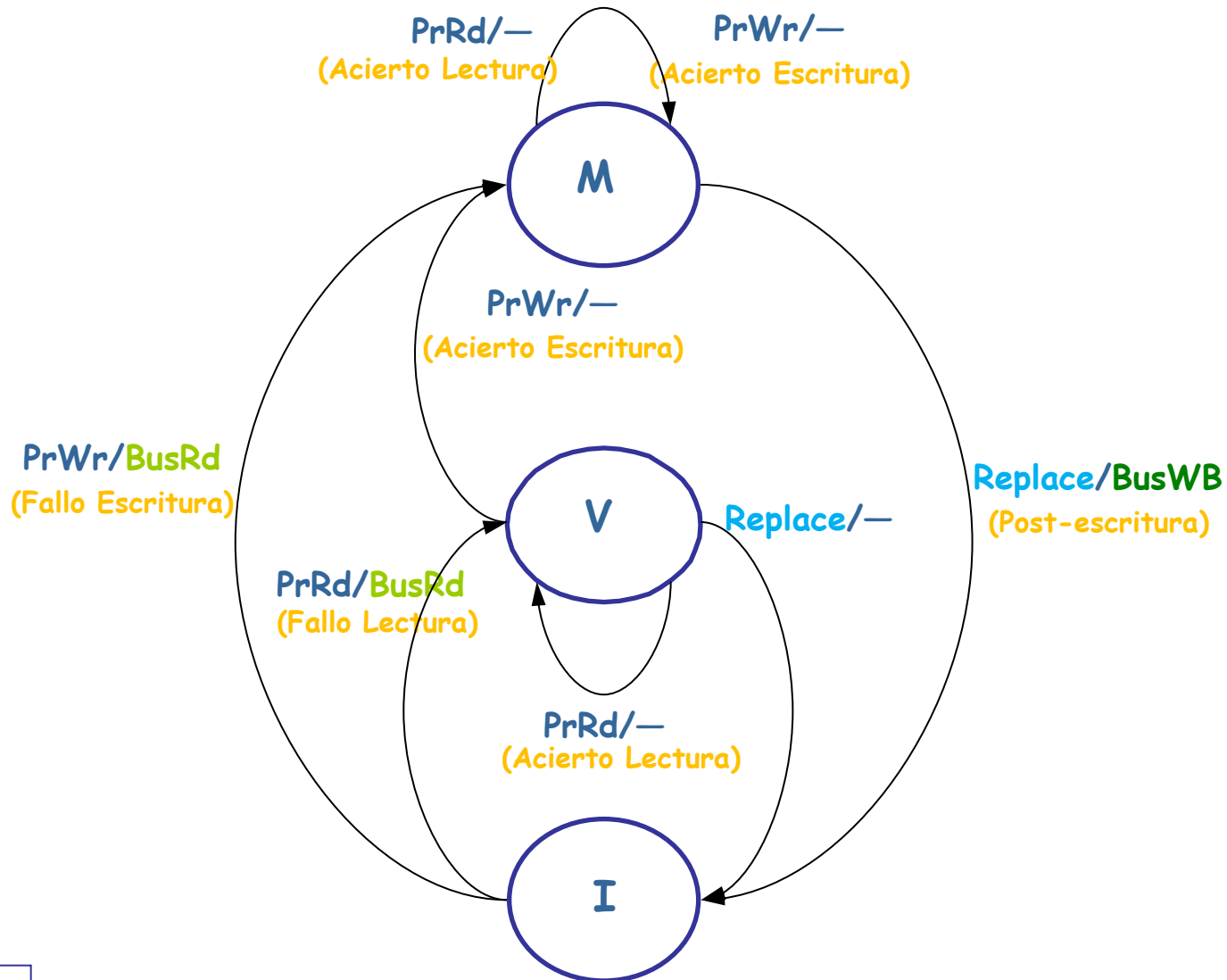
- ❑ Problema: Cada escritura hecha por cada procesador implica una transacción en el Bus → Consumo de Ancho de Banda
- ❑ Ejemplo:
  - o Procesador a 2000 MHz
  - o CPI = 1
  - o 15% stores de 8 bytes
  - o Se ejecutan 300 Millones de stores/s por procesador
    - $(0,15 \text{ stores/instr} \times 1 \text{ instr/ciclo} \times 2000 \times 10^6 \text{ ciclos/seg})$
  - o 2400 MB/s por procesador
  - o Un bus con un ancho de banda de 10GB/s sólo puede soportar 4 procesadores sin saturarse
- ❑ Evitar escrituras por medio de política write-back

# Recordatorio: caches write-back en un uniprocador

- ❑ Clave para conseguir un uso eficiente del ancho de banda (limitado) que proporciona un bus compartido:
  - o Post-Escritura (*write-back*)
  - o Asignación en Escritura (*write-allocate*)
  
- ❑ Diagrama de transición de estados de una cache con Post-Escritura (uniprocador)
  - o Tres estados para un bloque de memoria:
    - Invalido (o no presente)
    - Válido (*clean*)
    - Modificado (*dirty*)
  - o Dos tipos de accesos a memoria que implican transacciones de Bus
    - Lecturas (*BusRd*): provocadas por
      - Fallo de lectura
      - Fallo de escritura (recordar que hay asignación en escritura)
    - PostEscrituras (*BusWB*):
      - Reemplazamiento de bloque modificado

# Recordatorio: caches write-back en un uniprocador

- Diagrama de transición de estados de una cache con Post-Escritura



# Protocolo MSI: invalidación de 3 estados (multiprocesador)

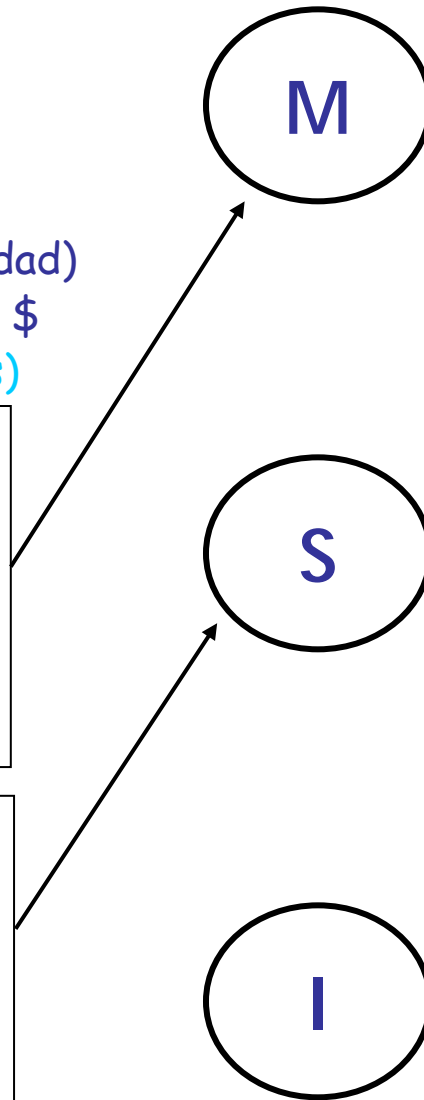
- ❑ Multiprocesador: Tres estados para un bloque de memoria
  - **M**odificado: Exclusivo
  - **S**hared (Compartido): Válido
  - **I**nválido
- ❑ Nueva transacción ordenada al Bus
  - BusRdx (Lectura con Exclusividad)
- ❑ Nueva respuesta del controlador de \$
  - **Flush** (volcar bloque de \$ a bus)

## Estado M:

- Sólo hay 1 cache con copia válida (la copia de la memoria principal está anticuada)
- La exclusividad implica que la cache puede modificar el bloque sin notificárselo a nadie

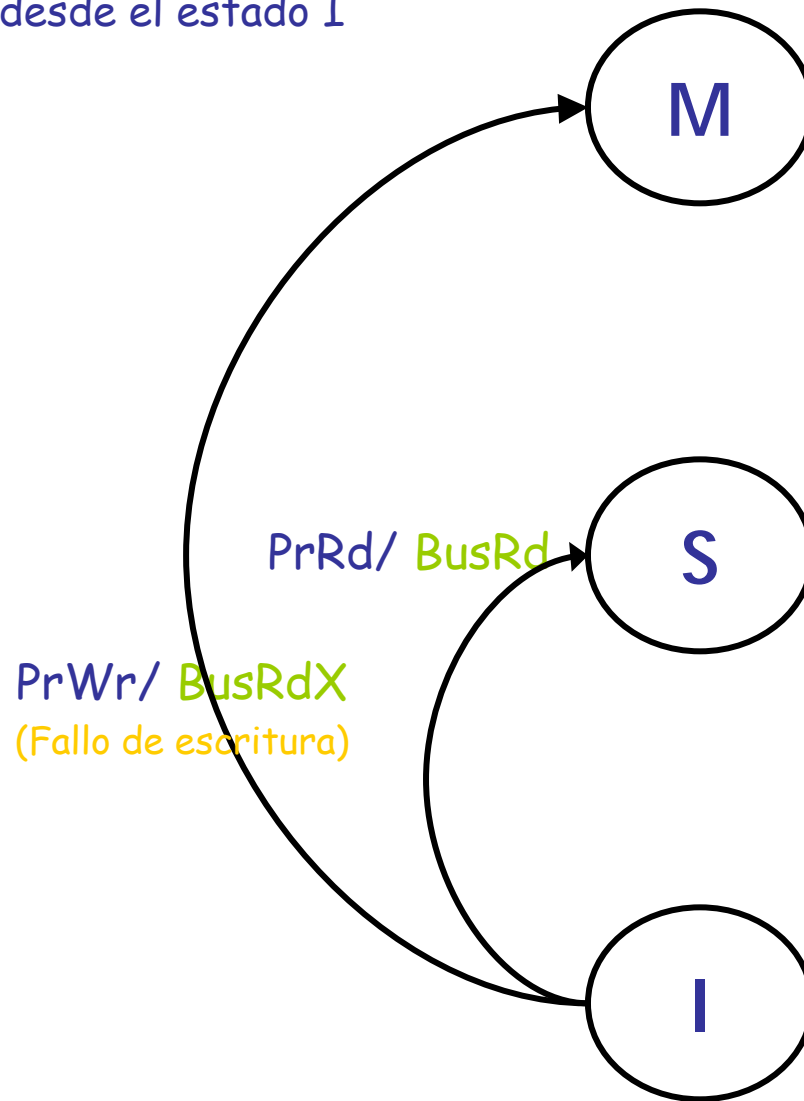
## Estado S:

- El bloque está presente en la cache y no ha sido modificado
- La memoria está actualizada
- Otras caches adicionales pueden tener copia



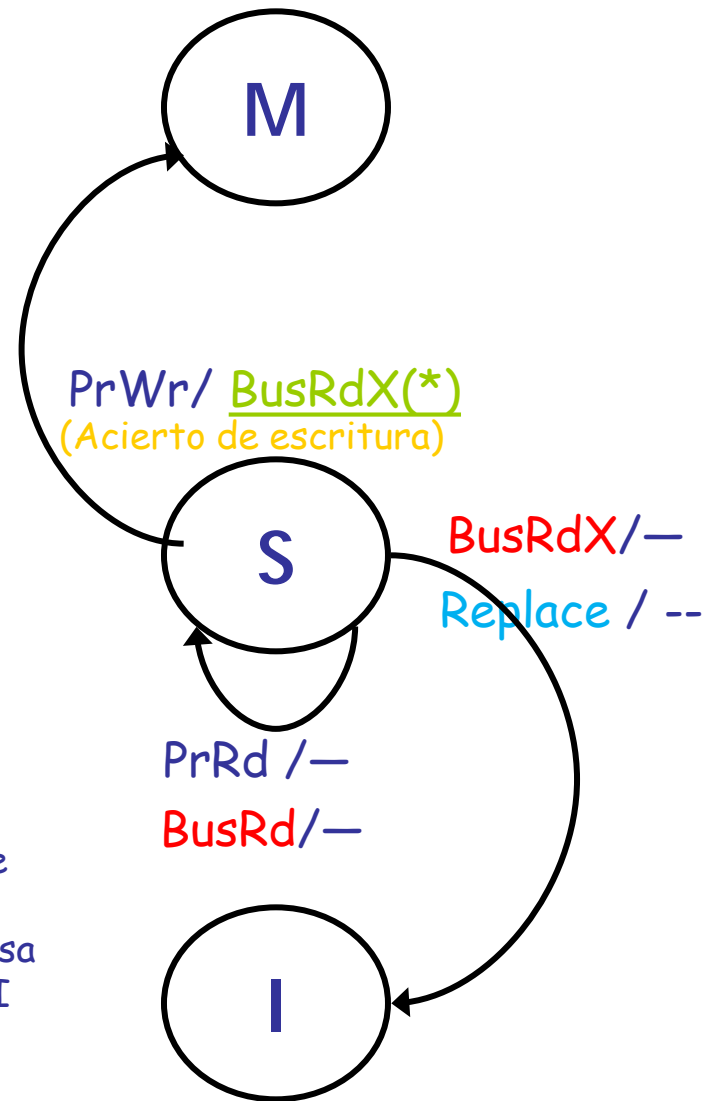
# Protocolo MSI: invalidación de 3 estados

❑ Transiciones desde el estado I



# Protocolo MSI: invalidación de 3 estados

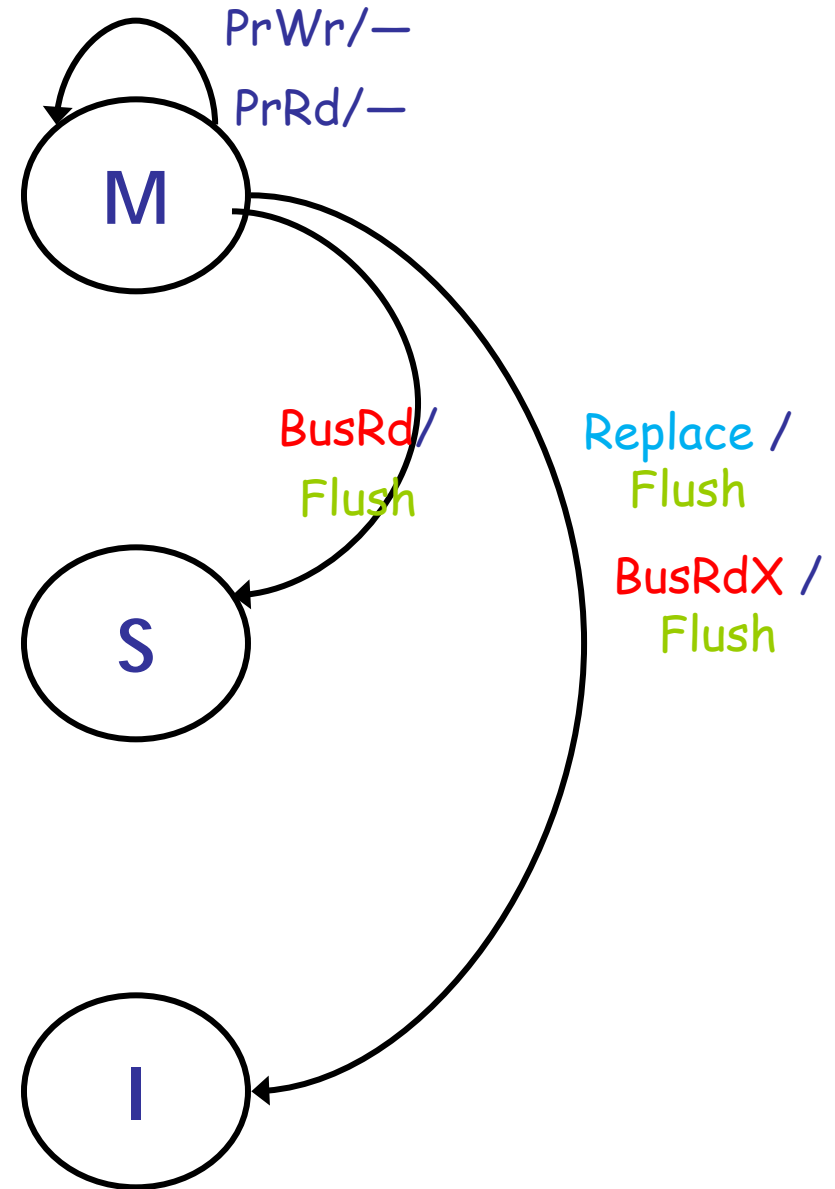
## ❑ Transiciones desde el estado S



(\*) Alternativa; Como la \$ ya tiene el bloque de datos también valdría una transacción llamada "**BusUpgrade**": Notificar al Bus que el bloque pasa a estado M para que las demás copias pasen a I

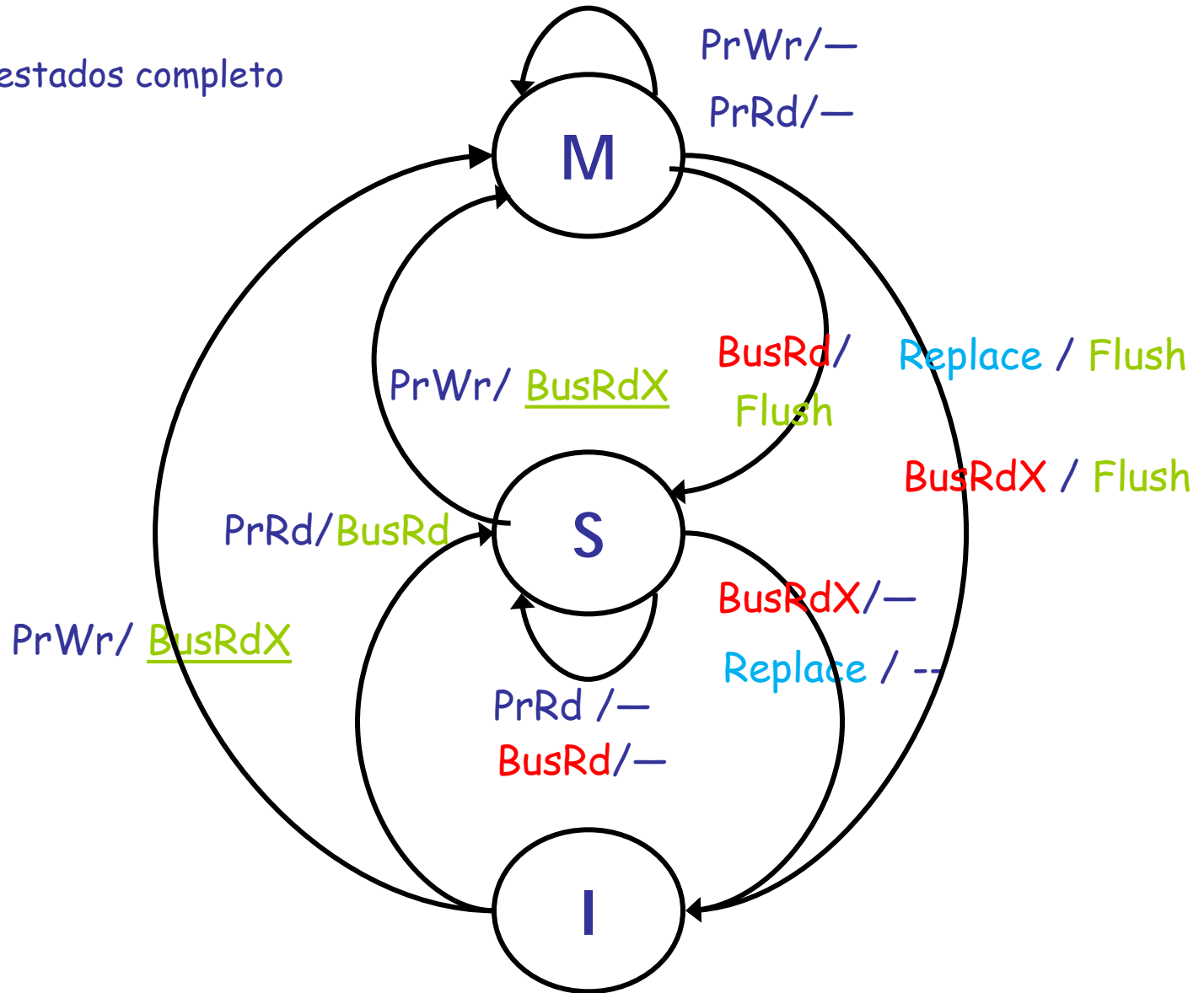
# Protocolo MSI: invalidación de 3 estados

❑ Transiciones desde el estado M

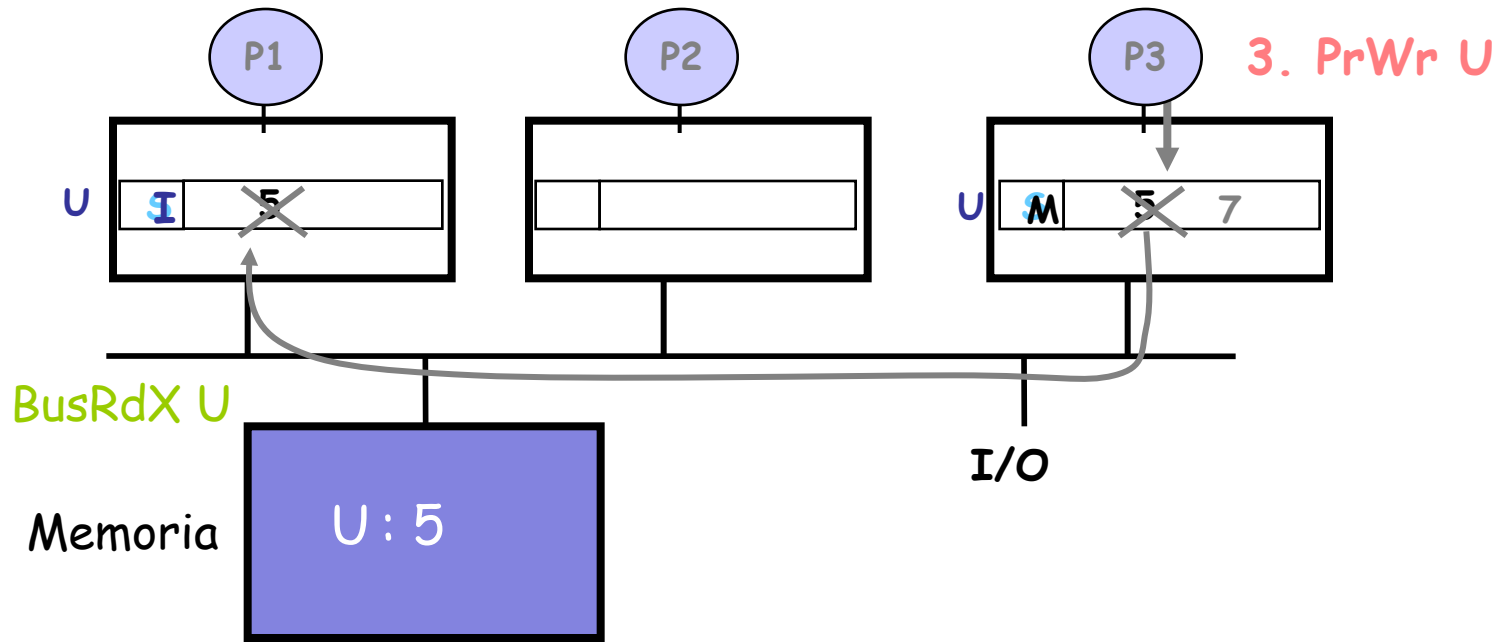


# Protocolo MSI: invalidación de 3 estados

□ Diagrama de estados completo

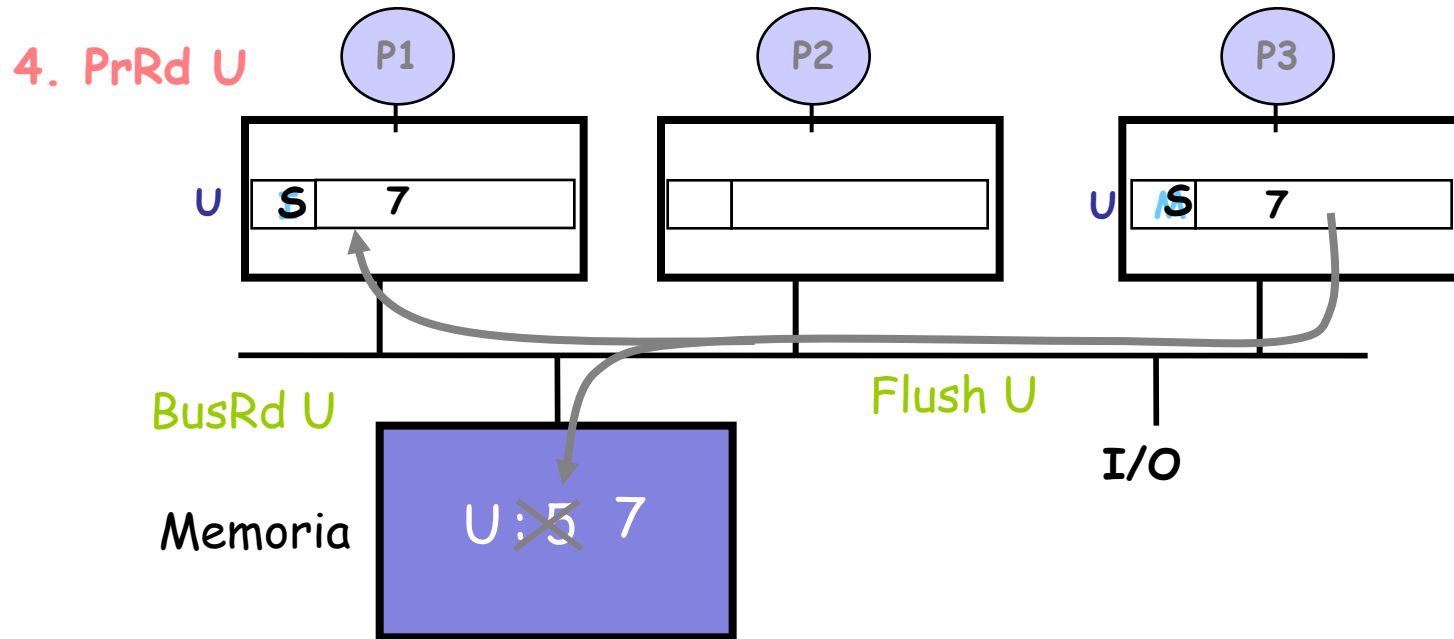


# Protocolo MSI: invalidación de 3 estados



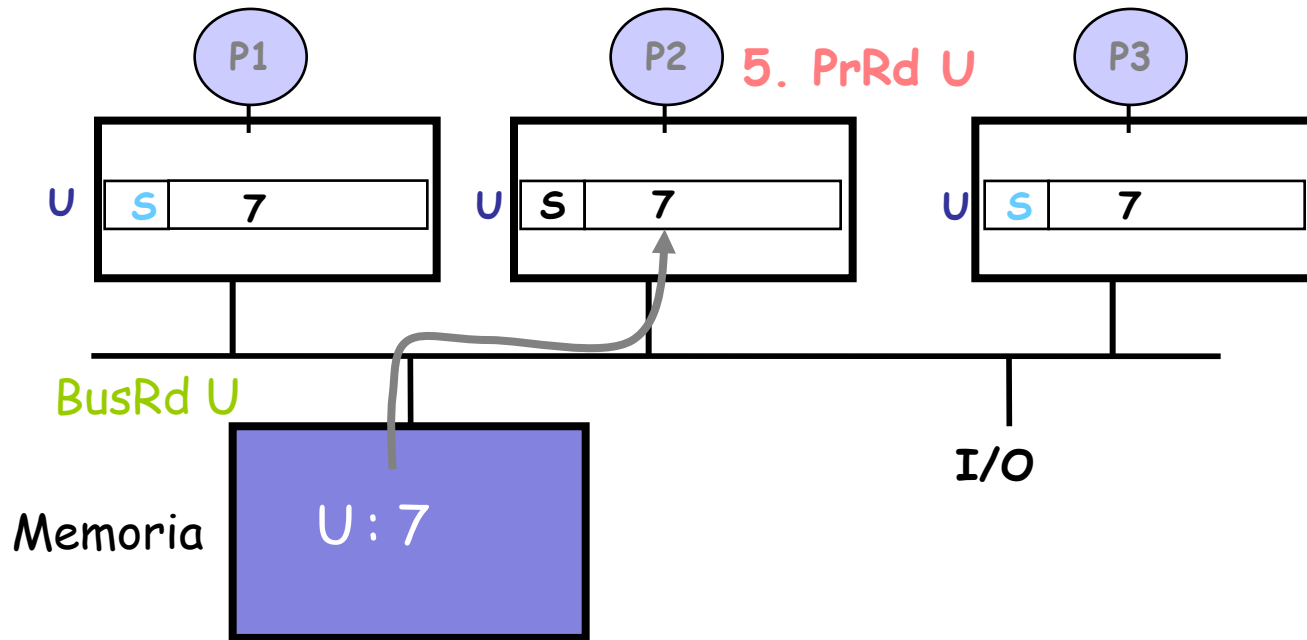
Operación	Estado P1	Estado P2	Estado P3	Transacción Bus	Datos Suministrados
P1 Lee U	S	—	—	BusRd	Memoria
P3 Lee U	S	—	S	BusRd	Memoria
P3 escribe U	I	—	M	BusRdX (o BusUpd)	Memoria (ignorados)
P1 Lee U					
P2 Lee U					

# Protocolo MSI: invalidación de 3 estados



Operación	Estado P1	Estado P2	Estado P3	Transacción Bus	Datos Suministrados
P1 Lee U	S	—	—	BusRd	Memoria
P3 Lee U	S	—	S	BusRd	Memoria
P3 escribe U	I	—	M	BusRdX (BusUpd)	Memoria (ignorados)
P1 Lee U	S	—	S	BusRd	Cache P3
P2 Lee U					

# Protocolo MSI: invalidación de 3 estados



Operación	Estado P1	Estado P2	Estado P3	Transacción Bus	Datos Suministrados
P1 Lee U	S	—	—	BusRd	Memoria
P3 Lee U	S	—	S	BusRd	Memoria
P3 escribe U	I	—	M	BusRdX (BusUpd)	Memoria (ignorados)
P1 Lee U	S	—	S	BusRd	Cache P3
P2 Lee U	S	S	S	BusRd	Memoria

# Implementación de protocolos *Snoopy*

---

- ❑ Carreras (races) entre escrituras
  - o No puede escribirse en la cache hasta que se tiene el bus
    - Otro procesador podría tomar el bus antes y escribir el mismo bloque en su cache
  - o Dos pasos:
    - Arbitrar el bus
    - Poner el fallo sobre el bus y completar la operación
- ❑ Los diferentes procesadores deben acceder al bus: datos y direcciones
- ❑ Los procesadores “espían” el bus:
  - o Si la dirección coincide con algún “tag” invalidar o actualizar
- ❑ Consulta de tag interfiere con uso de la cache por la CPU
  - o Solución : duplicar los tag
- ❑ El bus serializa las escrituras. Solo un procesador realiza la operación con memoria

# Protocolo MESI: invalidación de 4 estados

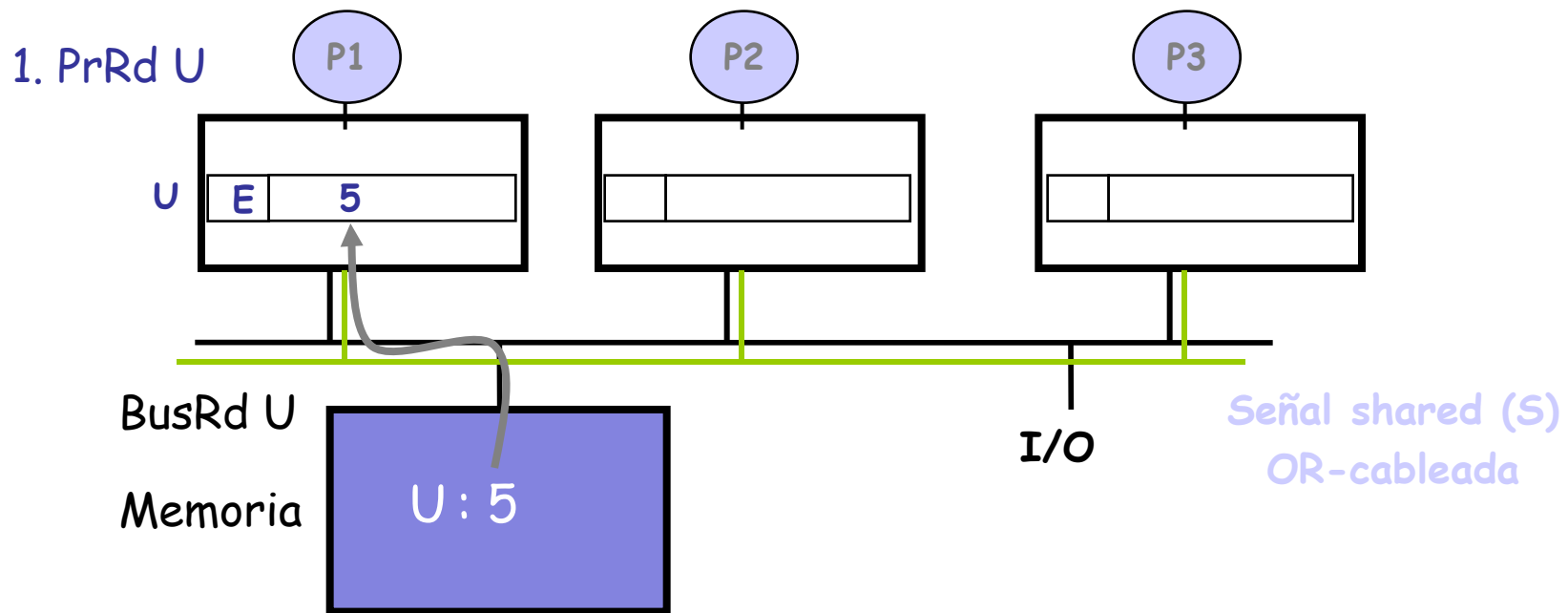
## ❑ Problema Protocolo MSI

- o Multiprogramación: carga de trabajo típica multiprocesadores pequeña escala
- o El protocolo MSI no se comporta bien con **Aplicaciones Secuenciales**
  - **Lectura-Modificación de un dato:** El protocolo MSI debe generar 2 transacciones de bus aunque no exista compartición
    - Al leer (I→S) seguida de al modificar (S→M)

## ❑ Protocolo MESI

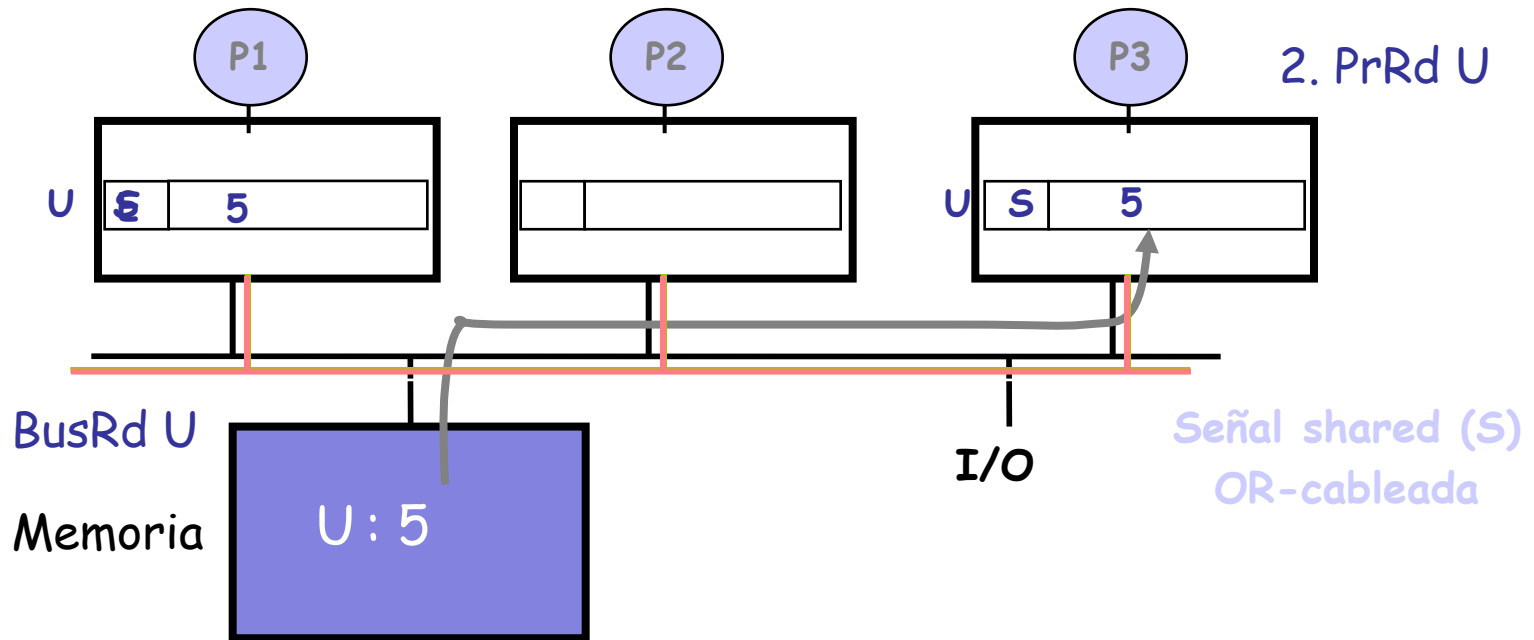
- o 4 Estados
  - (M) Modificado, (E) Exclusivo, (S) Compartido e (I) Inválido
  - Semántica similar a la del protocolo MSI
- o Nuevo estado (**E: Exclusive Clean o simplemente Exclusive**) indica que el bloque es la única copia pero no ha sido modificada: reduce tráfico en el bus
  - Nivel intermedio entre compartido y modificado
  - **Implica Exclusividad:** Puede pasarse a modificado sin realizar ninguna transacción (al contrario que Compartido)
  - **No implica pertenencia:** La memoria tiene una copia válida. El controlador de cache no debe proporcionar el bloque en respuesta a una transacción de lectura de dicho bloque (al contrario que Modificado)
- o **Señal S (Share).** En reposo si ninguna cache contiene el bloque que se ha solicitado: En fallo de lectura, leer bloque, y pasar a estado E.

# Protocolo MESI: invalidación de 4 estados



Operación	Estado P1	Estado P2	Estado P3	Transacción Bus	Datos Suministrados
P1 Lee U	E	—	—	BusRd	Memoria
P3 Lee U					
P3 escribe U					
P1 Lee U					
P2 Lee U					

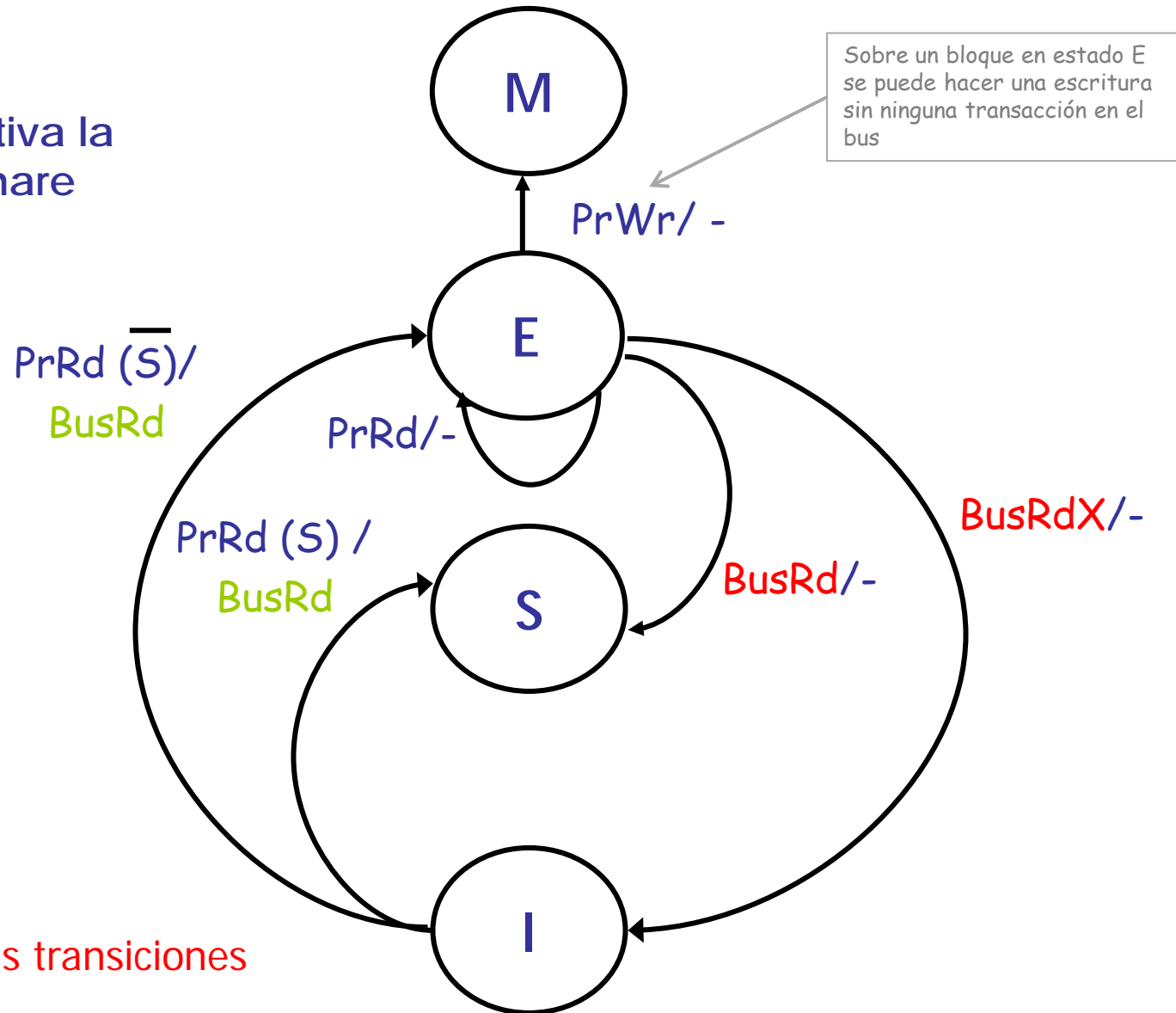
# Protocolo MESI: invalidación de 4 estados



Operación	Estado P1	Estado P2	Estado P3	Transacción Bus	Datos Suministrados
P1 Lee U	E	—	—	BusRd	Memoria
P3 Lee U	S	—	S	BusRd	Memoria
P3 escribe U					
P1 Lee U					
P2 Lee U					

# Protocolo MESI: invalidación de 4 estados

(S): se activa la señal share



# Protocolo MESI: invalidación de 4 estados

## ❑ Diagrama completo\* de un controlador de cache

Transiciones producidas por peticiones del procesador

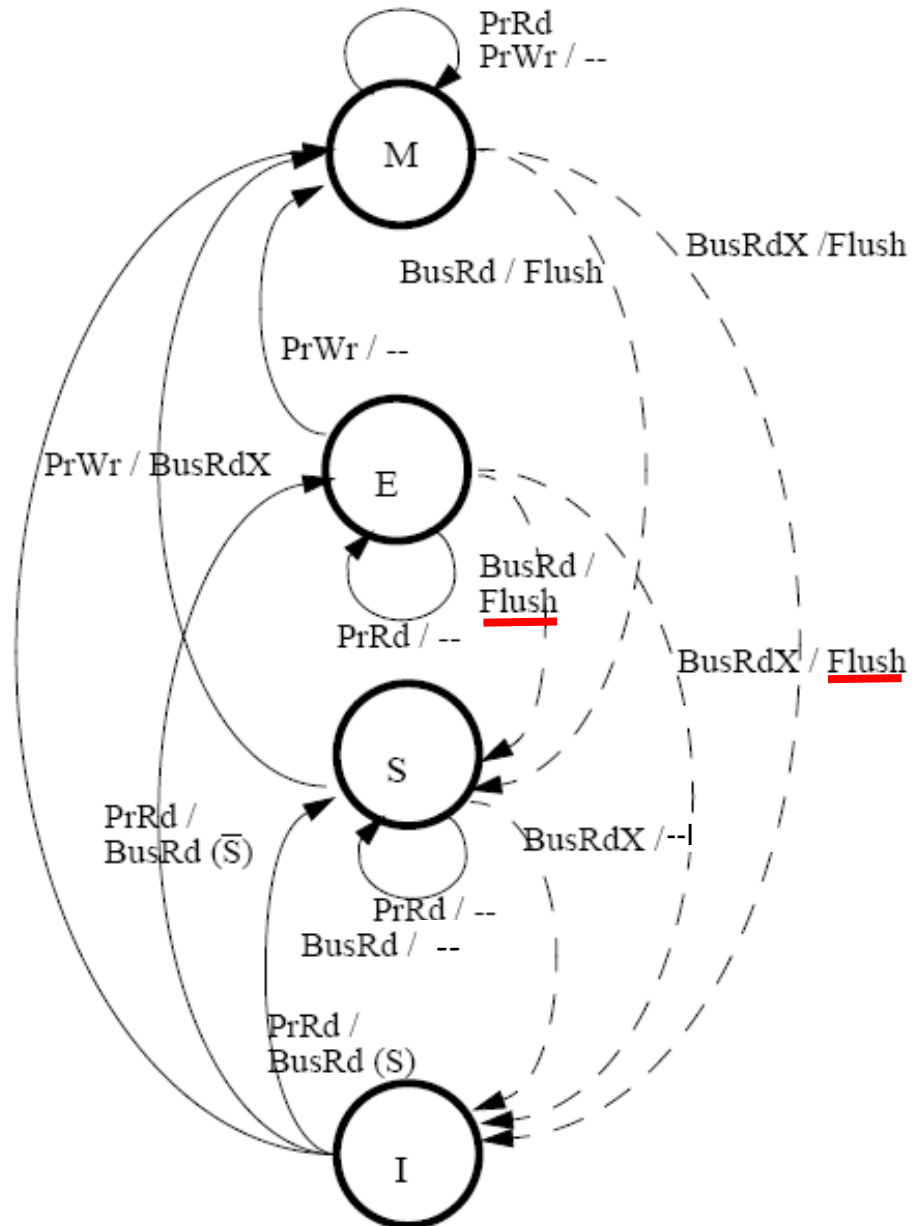


Transiciones producidas por eventos detectado en Bus



PrWr / BusRdX

(\*) No se incluyen las transiciones provocadas por reemplazamiento de bloques

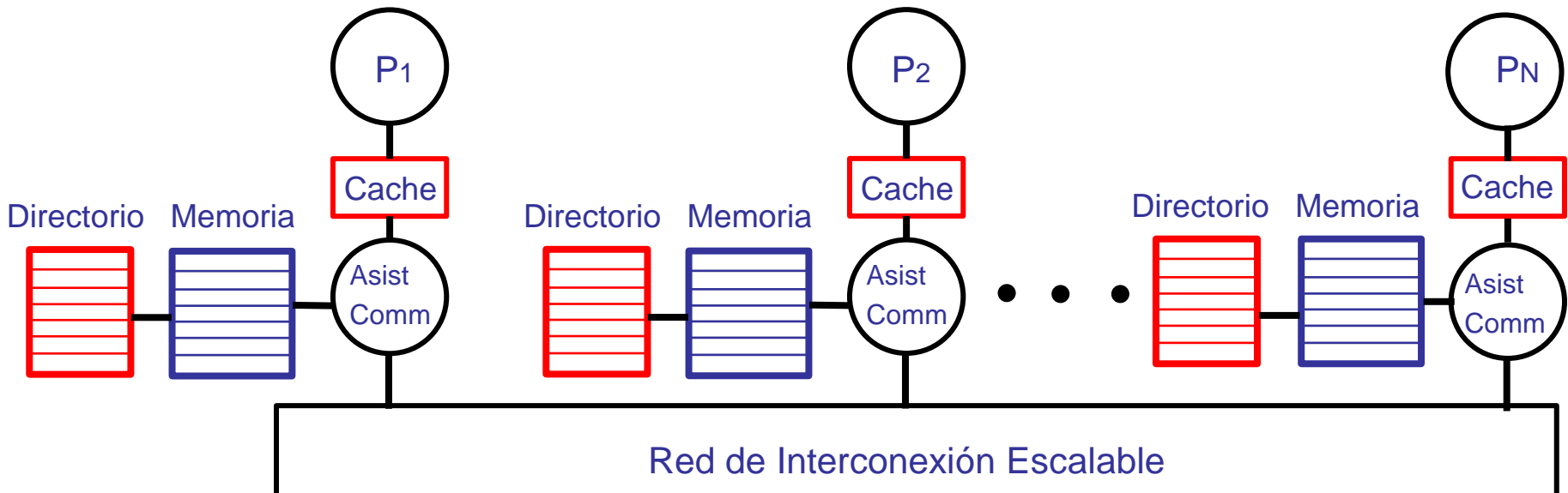


# Coherencia basada en directorio

## ❑ Sistemas de memoria distribuida (arquitectura NUMA)

Lectura: Culler 8.1, 2 y 3

- o Una zona de la memoria asociada a cada procesador
  - Pero... Cada procesador "ve" un espacio global de direcciones físicas compartido
- o Los accesos locales o remotos los maneja el Asistente de Comunicaciones del nodo.
  - El procesador y el controlador de cache, no perciben si un fallo de cache se resuelve en su memoria local o es preciso acudir a un nodo remoto de la red
- o Problema: coherencia de caches (igual que en snoopy).
  - Pero... **No hay un mecanismo para observar** las transacciones de los otros nodos
  - Por otra parte... es demasiado costoso mandar información (mensajes) a todos los demás nodos sobre las transacciones que hace cada uno.
  - Una solución: llevar a la cache solo datos locales (ejemplo: T3E)
- o Alternativa: Un directorio que mantiene información sobre el estado y localización en caches de cada bloque de memoria
  - El directorio podría ser un "cuello de botella": distribuir directorio.



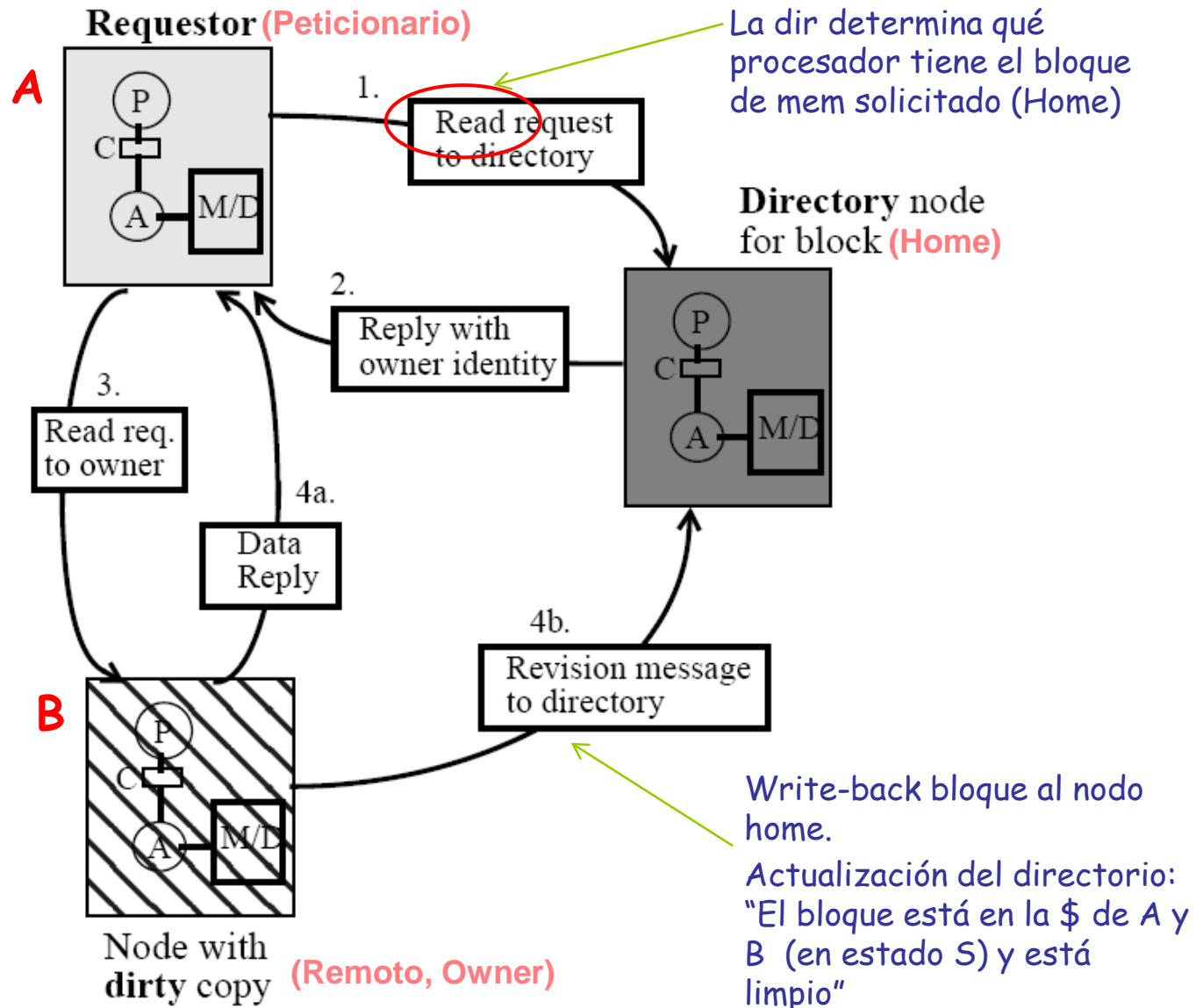
# Coherencia basada en directorio

- ❑ Protocolo de directorio: se pueden utilizar un esquema similar a snoopy (p. ej. con 3 estados: **MSI**)...
  - o ... pero los cambios de estado son provocados por mensajes a través de la red.
- ❑ Estados de un bloque en la cache de un procesador P
  - o **M**: Exclusivo (o Modificado).
    - La cache de P tiene la única copia válida del bloque. La Mp no está actualizada
  - o **S**: Compartido (Shared).
    - La cache de P tiene una copia del bloque. Puede haber más copias en otras caches. La Mp está actualizada.
  - o **I**: Inválido.
    - El bloque no está en la cache de P.

- ❑ El directorio permite determinar para cada bloque de memoria:
  - o Si el bloque no se encuentra copiado en la cache de ningún procesador
  - o Si existe una o varias copias del bloque sin modificar (clean, Mp actualizada) en los procesadores (**sharers**). Registra la lista procesadores que tienen esas copias.
  - o Si un procesador (y uno solo) tiene una copia modificada del bloque (dirty, Mp no actualizada) y cuál es ese procesador (**owner**).
  - o Posible implementación: Para cada bloque de Mp
    - 1 bit por procesador (indica presencia/ausencia del bloque en el procesador)
    - 1 bit clean/dirty
- ❑ Tres tipos de procesadores implicados en una transacción
  - o **Peticionario** (local): El procesador que inicia la petición
  - o **Home**: El procesador en donde reside la zona de memoria que contiene la dirección solicitada
  - o **Remoto**: Procesador cuya cache contiene copias válidas de la dirección solicitada en estado S o M. (Si estado es M → solo puede haber uno: Owner)

# Ejemplo de funcionamiento de un directorio

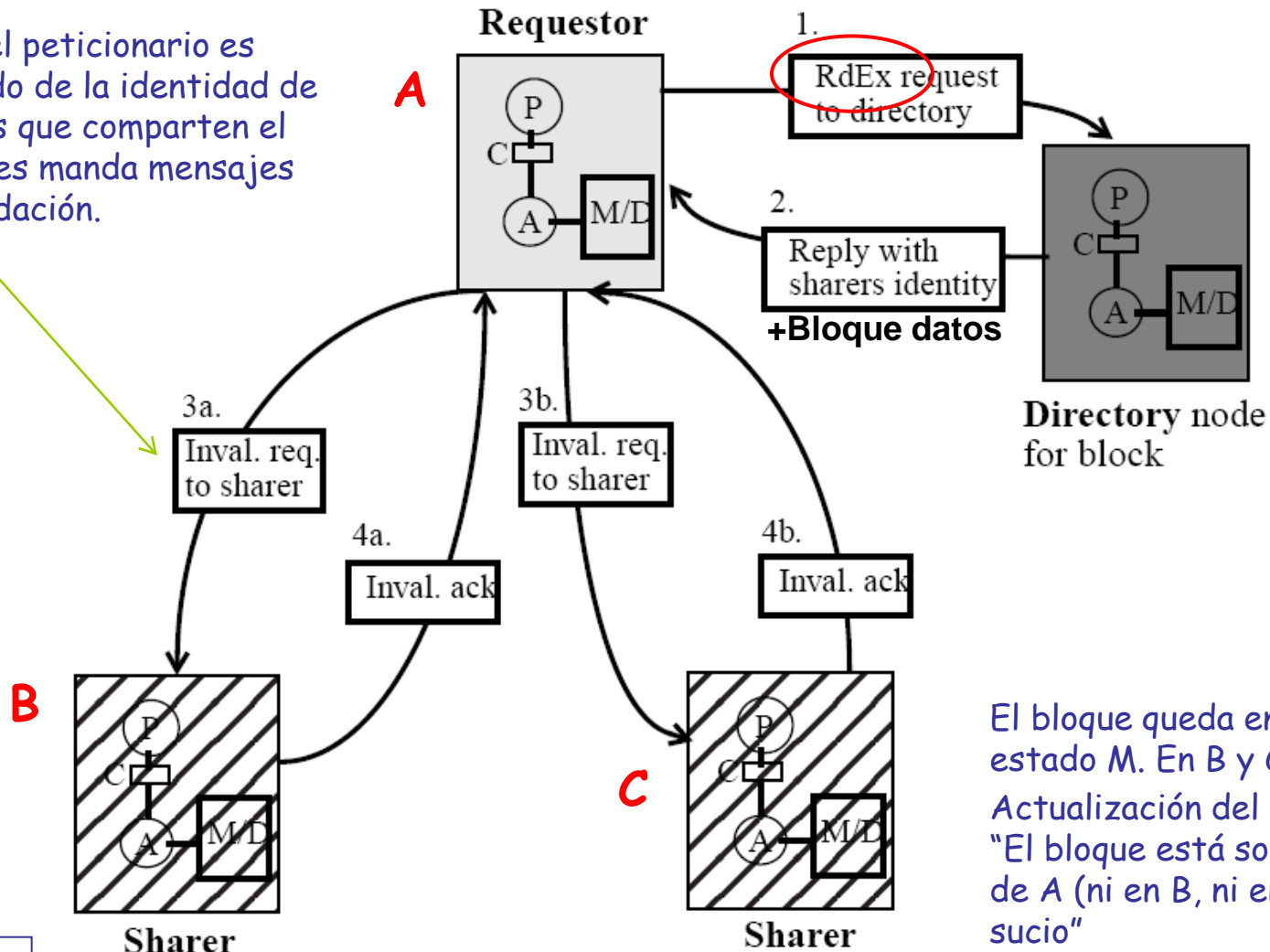
## ❑ Fallo de lectura sobre un bloque sucio



# Ejemplo de funcionamiento de un directorio

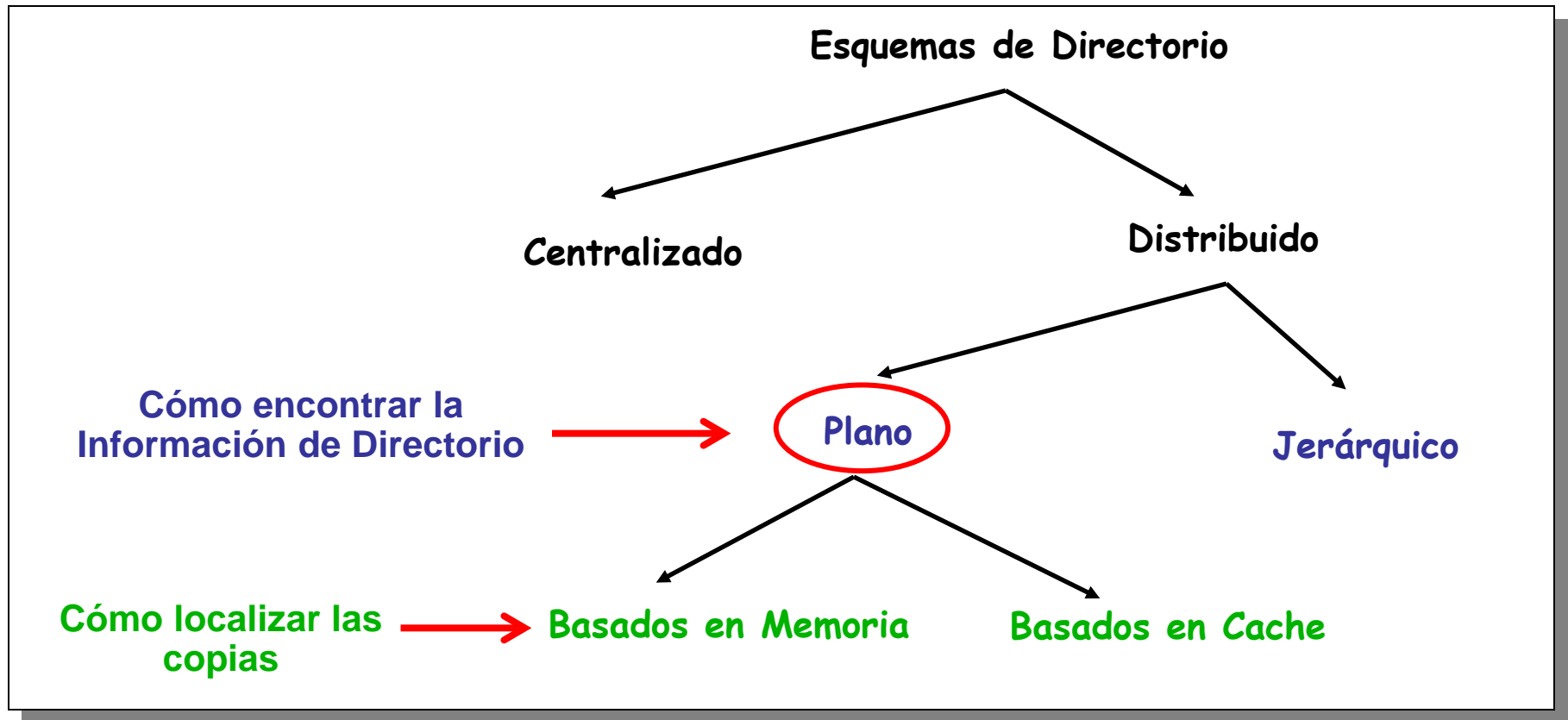
- ❑ Fallo de escritura sobre un bloque compartido por dos procesadores (bloque limpio)

Cuando el peticionario es informado de la identidad de los nodos que comparten el bloque, les manda mensajes de invalidación.



# Organización del directorio

- ❑ Primeros Esquemas de Directorio aparecieron en Mainframes de IBM. La memoria (centralizada) mantenía copia de las etiquetas (tags) almacenadas en cada una de las caches (Directorio Centralizado)



# Organización del directorio

---

## ❑ Esquema Plano

- o El **origen** de la información de directorio para un bloque se encuentra en un lugar fijo (nodo home/origen), que viene determinado directamente por la dirección del bloque (hashing)
- o Ante un fallo de cache se envía una transacción de red directamente al nodo origen para consultar el directorio

## ❑ Esquema Plano Basado en Memoria

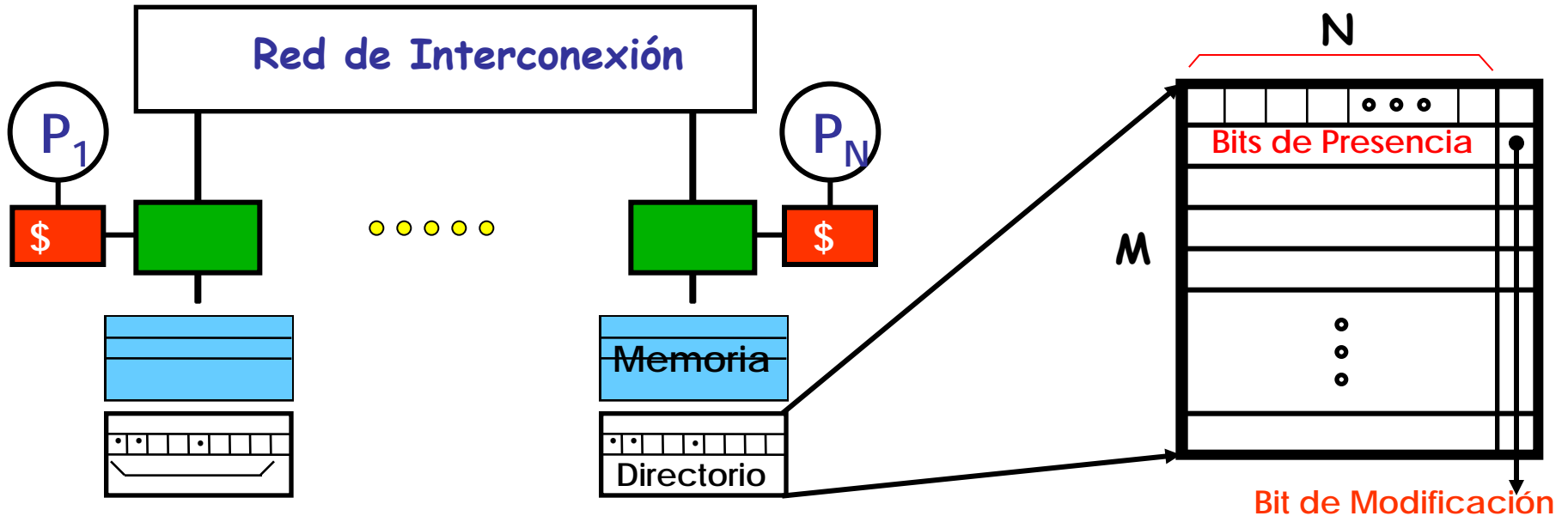
- o La información de directorio acerca de un determinado bloque esta almacenada en el nodo origen (home) de dicho bloque (Stanford DASH/FLASH, SGI Origin, MIT Alewife, HAL)

## ❑ Esquema Plano Basado en Cache

- o La información de directorio acerca de un determinado bloque no está totalmente almacenada en el nodo origen sino distribuido entre las propias copias (IEEE SCI, Sequent NUMA-Q)

# Directorio plano basado en memoria

- $N$  nodos,  $M$  bloques de memoria por nodo.



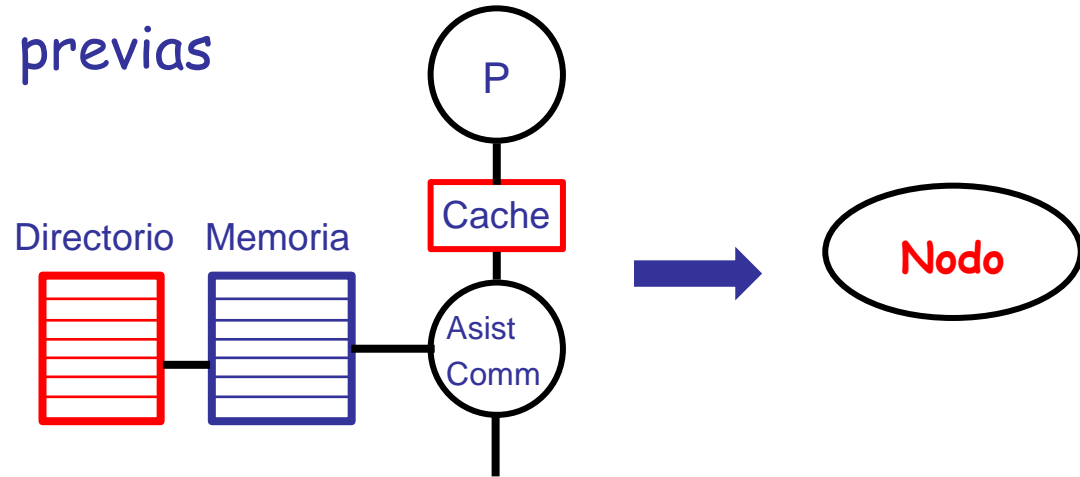
## Organización más natural: Full Bit Vector

- 1 bit de presencia por nodo (indican en qué nodos está copiado cada bloque de  $M_p$ )
- Estado: uno o más bits
  - No es necesario saber estado concreto del protocolo
  - Más simple: un único bit (bit de modificación)

# Directorio plano basado en memoria: funcionamiento

## □ Definiciones previas

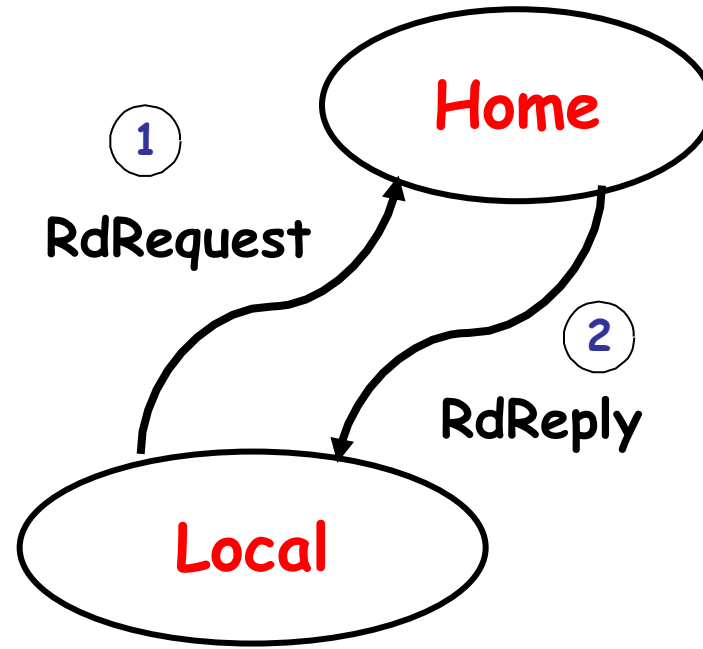
### o Nodo



- o Nodo Peticionario (local): contiene el procesador que emite una petición del bloque
- o Nodo Origen (home): nodo en cuya porción de memoria principal se encuentra el bloque (contiene el directorio)
- o Nodo Sucio (dirty): tiene una copia del bloque en su cache en estado modificado
- o Nodo Exclusivo (exclusive): tiene una copia del bloque en estado exclusivo (única copia de cache válida y el bloque esta actualizado en memoria)
- o Nodo Propietario (owner): nodo que mantiene en la actualidad la copia válida de un bloque y debe proporcionar los datos cuando sean necesarios (Sucio o Origen)

# Directorio plano basado en memoria: funcionamiento

- ❑ Obtener bloque para lectura (fallo de lectura)
  - o Caso 1: Dirty bit OFF (el bloque está actualizado en la memoria del nodo Home)
    - 1. El nodo local pide el bloque
    - 2. El Home lo proporciona (y actualiza el directorio)
    - Sólo son necesarias dos transacciones en la red

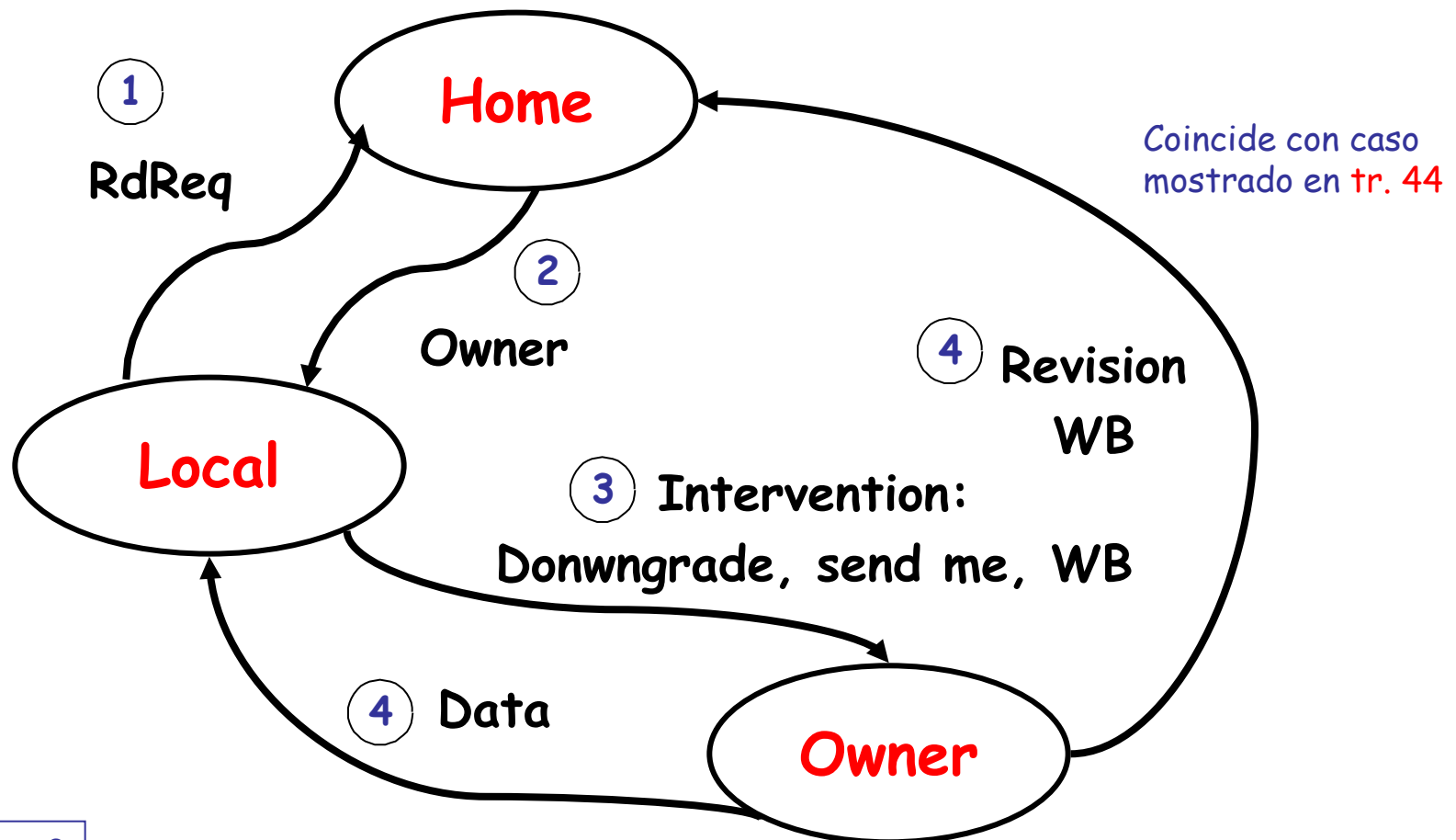


# Directorio plano basado en memoria: funcionamiento

## ❑ Obtener bloque para lectura (fallo de lectura)

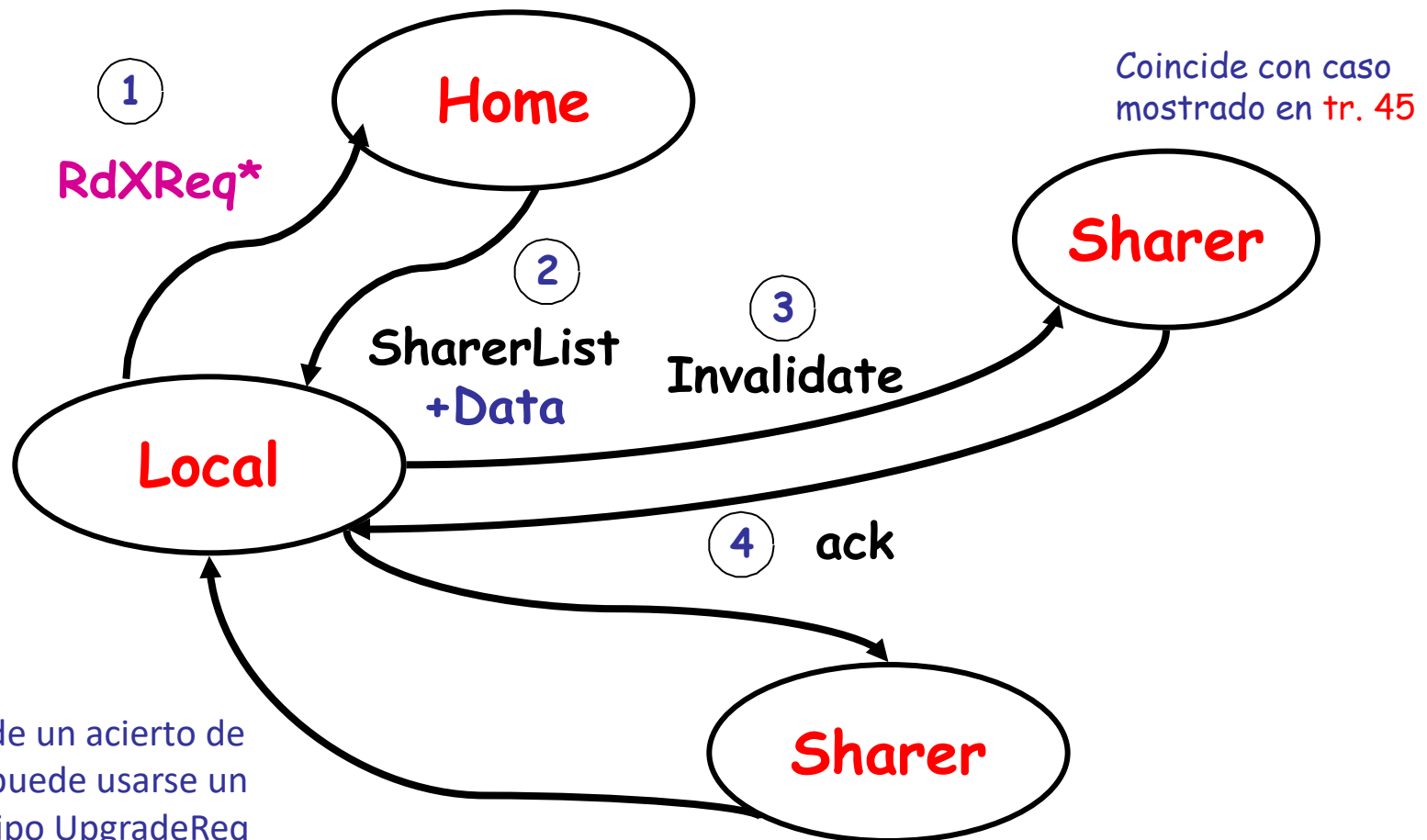
o Caso 2: Dirty bit ON (El bloque no está actualizado en memoria del nodo Home)

- 1. El nodo local pide el bloque
- 2. El Home proporciona la identidad del nodo Propietario
- 3. El nodo local hace una petición de lectura al propietario
- 4. El nodo propietario proporciona el bloque y hace WB al Home



# Directorio plano basado en memoria: funcionamiento

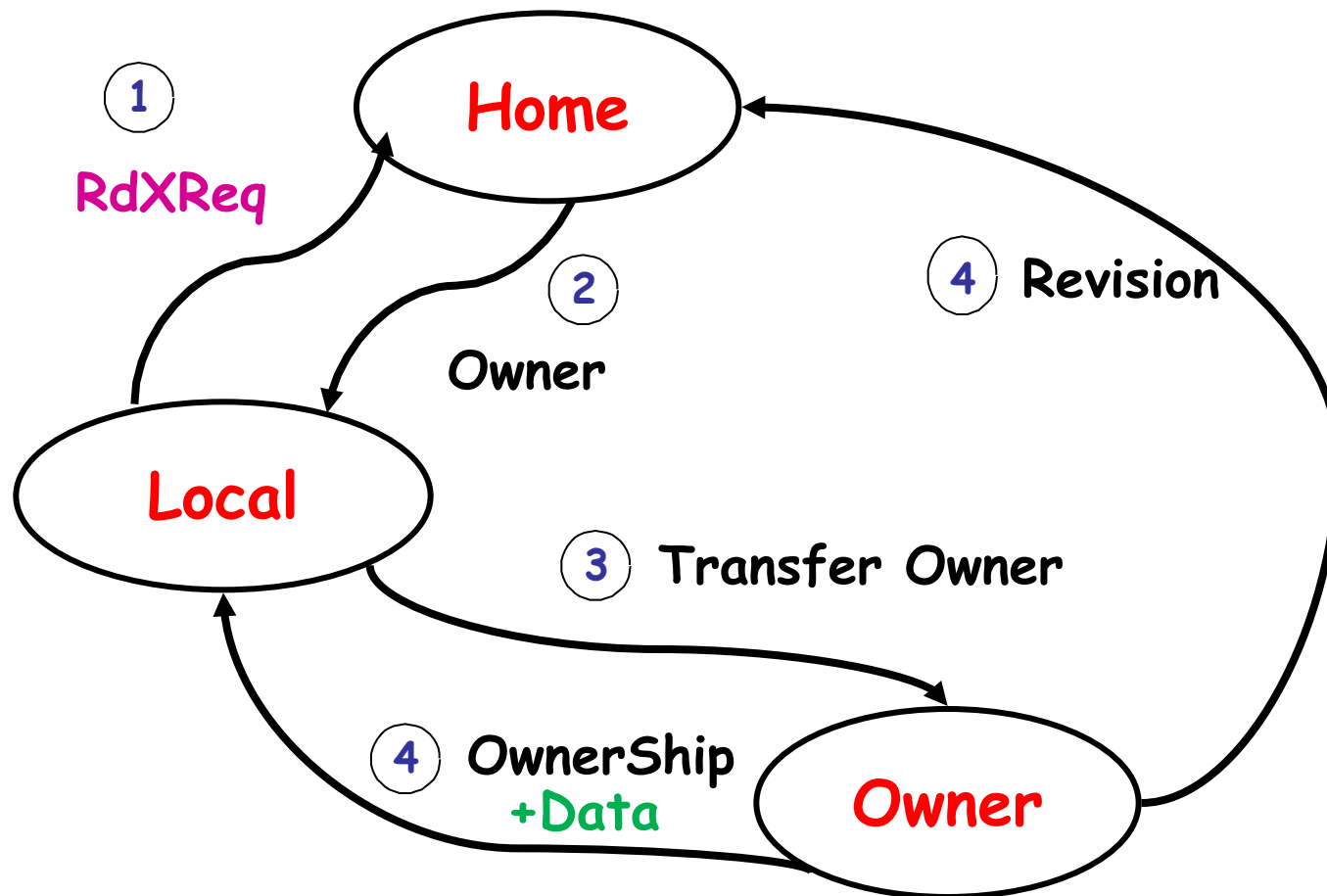
- ❑ Obtener bloque para escritura (**fallo o acierto escr.**)
  - o Caso 1: Dirty bit OFF (hay varias copias válidas. Mp actualizada)



\*En caso de un acierto de escritura puede usarse un mensaje tipo UpgradeReq

# Directorio plano basado en memoria: funcionamiento

- ❑ Obtener bloque para escritura (**fallo escritura**)
  - o Caso 2: Dirty bit ON (hay una copia válida en el propietario. Mp no actualizada)

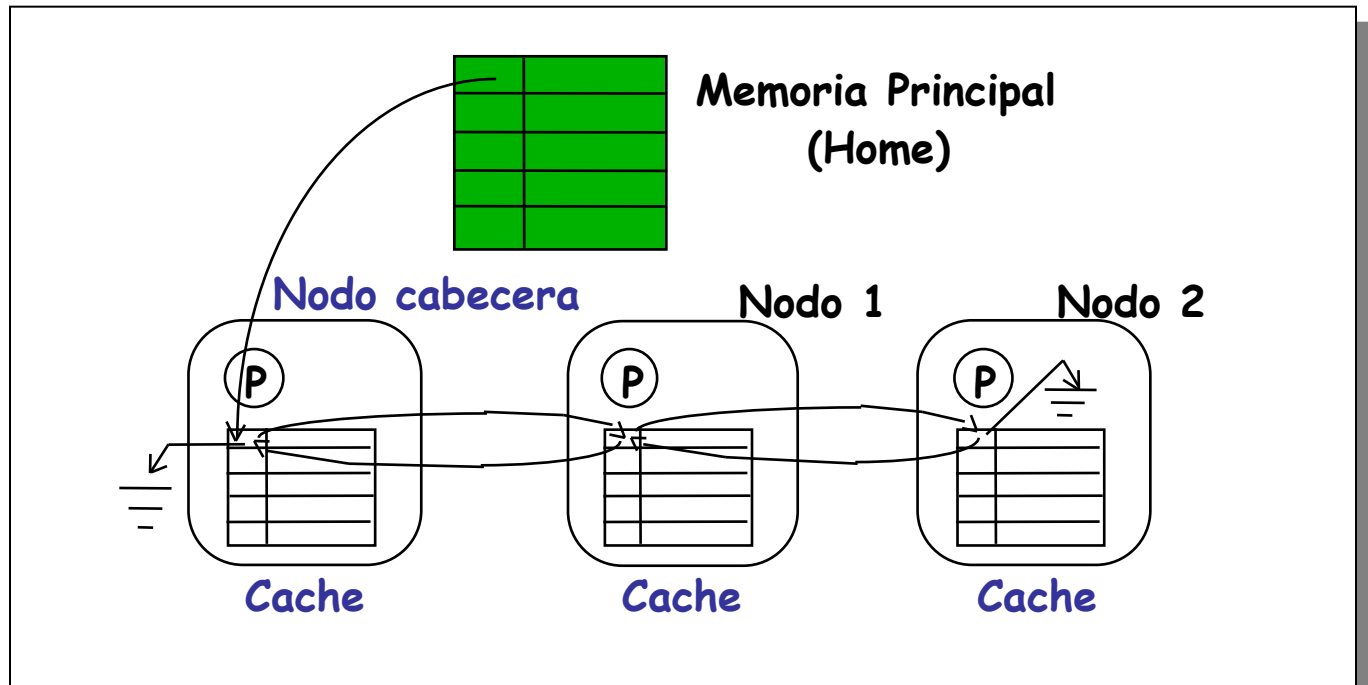


# Directorio plano basado en memoria

- ❑ Rendimiento en las Escrituras (mensajes de invalidación)
  - o Número de mensajes (BW): proporcional al número de nodos que comparten el bloque
  - o Número de mensajes en el camino crítico (Latencia): La identidad de todos ellos está disponible en el nodo origen, los mensajes de invalidación se pueden enviar en paralelo
- ❑ Sobrecarga de almacenamiento:
  - o  $M$  bloques de memoria : Sobrecarga proporcional a  $N_{proc} * M$
  - o Suponiendo tamaño de línea = 64 bytes
    - Con 64 nodos: sobrecarga del directorio 12.5%
    - Con 256 nodos: sobrecarga del directorio 50%
    - Con 1024 nodos: sobrecarga del directorio 200%
- ❑ Soluciones:
  - o Utilizar nodos multiprocesador (solo 1 bit por nodo) Ejemplo: 256 procesadores, 4 procesadores por nodo, línea de 128 B : Sobrecarga: 6.25%
  - o En lugar de almacenar 1 bit por nodo en cada entrada del directorio, se almacenan un determinado número de punteros,  $N_{punt}$ , como máximo, a los nodos que comparten dicho bloque (reducción de anchura de dir)
    - Ejemplo: Para 1024 nodos. 1024 bits vs.  $N_{punt} * 10$  bits
  - o Organizar el directorio como una cache, en lugar de tener una entrada por cada bloque (reducción de altura del dir)

# Directorio plano basado en cache

- ❑ Existe una memoria principal origen de cada bloque
  - o La entrada del directorio en el nodo origen tiene:
    - Un puntero al primero de los nodos (nodo cabecera) que tiene copia de dicho bloque
    - Bits de estado
  - o El resto de los nodos con copia se encuentra unidos mediante una lista enlazada distribuida (IEEE 1596-1992 Scalable Coherent Interface (SCI) : lista doblemente enlazada)



# Directorio plano basado en cache

---

## ❑ Fallo de lectura:

- o El nodo peticionario (**local**) envía una petición al nodo origen de ese bloque para determinar la identidad del nodo cabecera
- o El nodo origen (**home**) responde con la identidad del nodo cabecera (**si existe**)
- o El nodo peticionario (**local**) envía solicitud al nodo cabecera para que le inserte a la cabeza de la lista (se convierte en el nuevo nodo cabecera) : incrementa la latencia
- o Los datos son enviado por
  - Nodo origen (**home**) si tiene copia
  - Nodo cabecera en caso contrario (siempre tiene la última copia)

## ❑ Eliminación (Reemplazamiento) e inserción en la lista

- o Añade bastante complejidad al protocolo: es necesario llevar a cabo coordinación con el nodo anterior y posterior en la lista (podría estar intentando reemplazar el mismo bloque)

## ❑ Escritura:

- o El nodo peticionario (**local**) puede encontrarse ya en la lista de nodos que comparten el bloque
- o Se recorre el resto de la lista para invalidar las sucesivas copias.
  - Los acuse de recibo de las invalidaciones se envían al nodo que realiza la escritura

## ❑ Rendimiento en las Escrituras (mensajes de invalidación)

- o Número de mensajes (BW): proporcional al número de nodos que comparten el bloque (semejante a los basados en memoria), pero están distribuidos
- o Número de mensajes en el camino crítico (Latencia): también es proporcional al número de nodos que comparten el bloque (pero los mensajes se serializan)

- ❑ Ventajas respecto a los esquemas basados en memoria
  - o Menor sobrecarga de directorio.
    - Punteros a nodo cabecera:
      - Proporcional al número de bloques de memoria de la máquina
    - Punteros siguiente y anterior:
      - Proporcional al número de bloques de cache en la máquina (mucho menor que el número de bloques de memoria)
  - o El trabajo realizado por los controladores para enviar las invalidaciones no está centralizado en el nodo origen sino distribuido entre los nodos que comparten el bloque
  - o La lista enlazada guarda el orden en el que se han realizado los accesos

- ❑ Necesidad de garantizar acceso seguro de un proceso a variables compartidas
- ❑ Problema que aparece ya en uniprosesadores con multiprogramación (exclusión mutua, secciones críticas)
- ❑ Aspectos relevantes
  - o Primitiva hw básica: una instrucción no interrumpible (operación atómica) que lee y actualiza una cierta posición de memoria (cerrojo).
  - o Las operaciones de sincronización a nivel de usuario (rutinas sw de sincronización) se construyen usando esta primitiva hw.
  - o Para sistemas MPP las operaciones de sincronización pueden ser el cuello de botella del sistema
    - Hay un conjunto de procesos compitiendo por leer-y-modificar el cerrojo (adquisición del cerrojo).
    - A diferencia del caso uniprosesador, todos estos procesos pueden estar corriendo simultáneamente en los diferentes procesadores.
    - Cada procesador puede hacer las lecturas del cerrojo sobre su cache local, sin generar tráfico en el bus.
    - Pero... al escribir genera tráfico en el sistema de comunicaciones (invalidaciones) incluso aunque el intento de modificar el cerrojo sea fallido porque el cerrojo ya haya sido adquirido por otro procesador.
    - El procesador puede permanecer en un bucle generando tráfico inútil en el bus hasta que logra adquirir el cerrojo.

# Sincronización: Primitivas hw básicas

- ❑ Intercambio (atomic exchange): Intercambia el valor de un registro y una posición de memoria (cerrojo).
  - o Convenio de interpretación de los valores del cerrojo
    - 0 => el cerrojo está libre
    - 1 => el cerrojo está bloqueado y no es accesible.

SI situación inicial:

Reg 1    Mem 0



Atomic  
Exchange

Reg 0    Mem 1

al observar que Reg=0 tengo la confirmación de que el cerrojo estaba libre y lo he bloqueado

SI situación inicial:

Reg 1    Mem 1



Atomic  
Exchange

Reg 1    Mem 1

al observar que Reg=1 tengo la confirmación de que el cerrojo estaba bloqueado por otro proceso y no lo he cambiado

- o Si dos procesadores intentaran ejecutar el Exchange a la vez, el mecanismo de **serialización de escrituras** garantiza que uno de los dos obtendrá un 0 en el Reg y el otro obtendrá un 1.

# Sincronización: Primitivas hw básicas

## ❑ Otras primitivas

- o Test-and-set. Comprueba el valor de una variable y si está a cero, la pone a 1.
- o Fetch-and-increment. Incrementa el valor de una variable en memoria.

## ❑ Operaciones atómicas en multiprocesadores

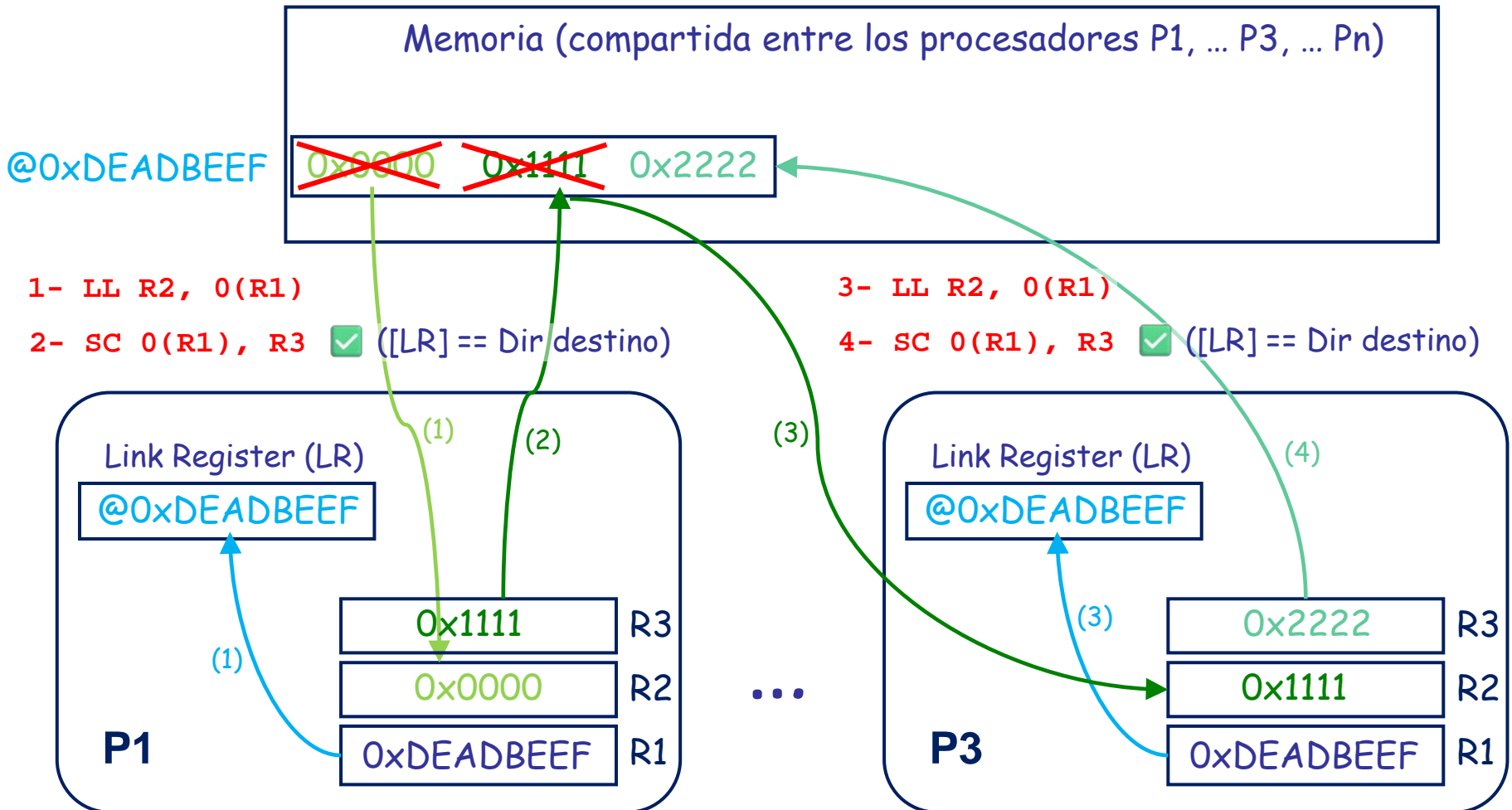
- o Se requiere una lectura y una escritura en una sola instrucción no interrumpible
- o Complica la implementación de la coherencia: el hw no puede permitir otra op entre la lectura y la escritura, y además debe evitar el interbloqueo !!

## ❑ Sincronización con un par de operaciones

- o **Load Linked (LL)**: Lee una dir de memoria, pero además guarda en un registro especial (link register) la dirección leída.
  - Si ocurre una **interrupción** (cambio de contexto), o si el bloque de cache que contiene la dir es invalidado por una **escritura**, el link register se borra
- o **Store Conditional (SC)**: Almacena el contenido de un registro R en una dir de memoria, siempre que ésta coincida con la dir almacenada en el link register. En caso contrario no se hace el almacenamiento.
- o Si el almacenamiento tuvo éxito devuelve  $R=1$  y en caso contrario  $R=0$ . Analizando el valor retornado por SC se puede deducir si el par se ejecutó como si las instrucciones fueran atómicas

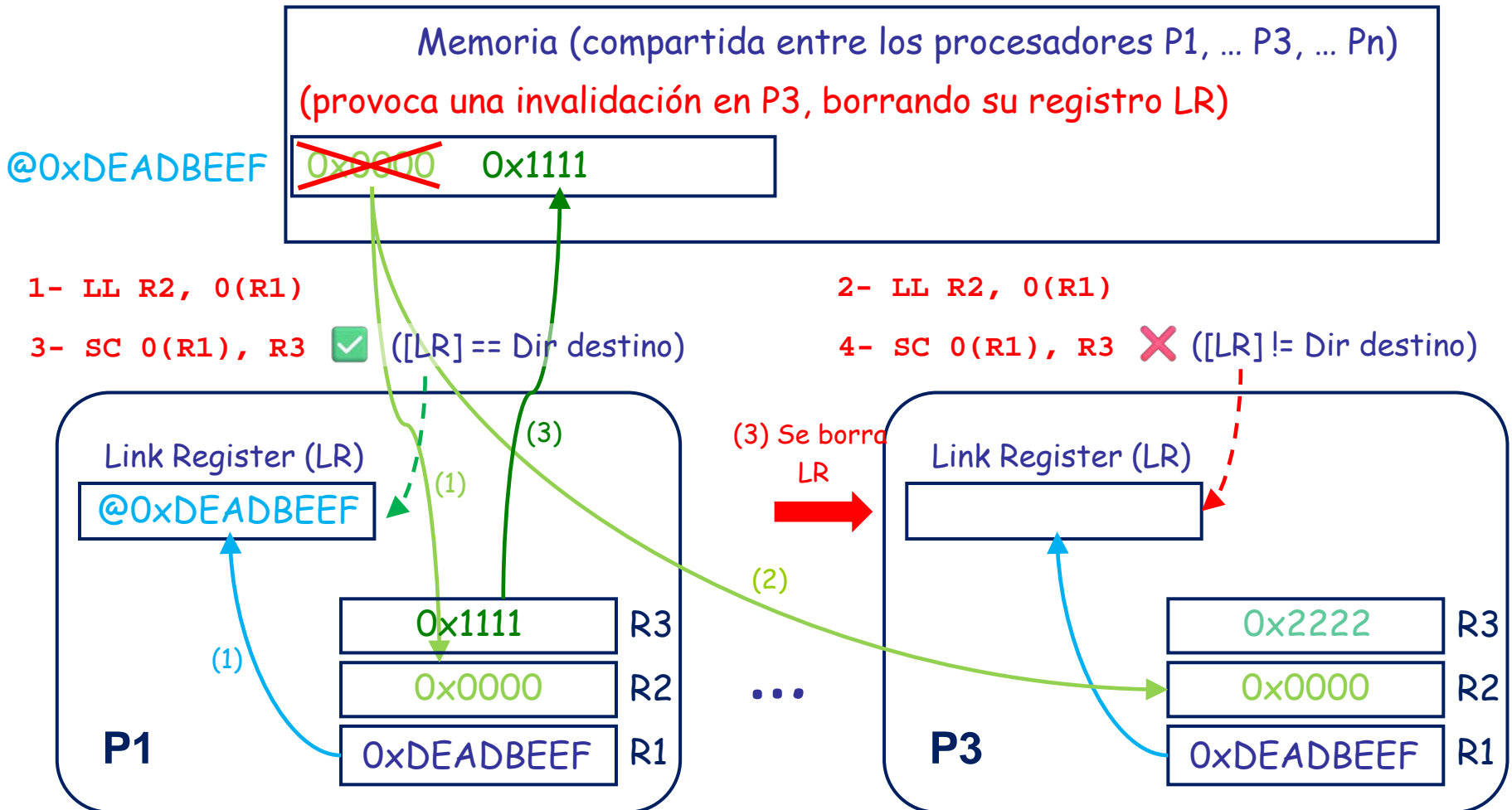
# Load Linked (LL) + Store Conditional (SC)

1. Operación Exchange usando LL+SC: P1 y P3 no se molestan entre sí: ambos tienen éxito



# Load Linked (LL) + Store Conditional (SC)

2. Operación Exchange usando LL+SC: P1 y P3 interfieren entre ellos: P1 tiene éxito, P3 no



# Sincronización: Primitivas hw básicas

❑ Implementación de otras primitivas con LL+SC

❑ Exchange. Ej.: intercambio  $R4 \leftrightarrow M[0+(R1)]$

```
try:      MOV      R3,R4          ;move exchange value to R3
          LL        R2,0(R1)      ;load linked
          SC        R3,0(R1)      ;store conditional
          BEQZR3,try              ;branch if store fails
          MOV      R4,R2          ;success: put load value in R4
```

❑ Fetch-and-increment. Ej.: Incrementa atómicamente  $M[0+(R1)]$

```
try:      LL        R2,0(R1)      ;load linked
          DADDUI    R3,R2,#1      ;increment (*)
          SC        R3,0(R1)      ;store conditional
          BEQZ      R3,try        ;branch if store fails
```

(\*) Entre LL y SC puede haber op sobre registros; problemas potenciales en caso contrario

# Implementación de spin locks

- ❑ Spin lock: bucle en el que un proceso permanece intentando conseguir un cerrojo hasta que lo consigue.  
**PROBLEMA: Si varios procesadores → Aumento de tráfico!!**
- ❑ Implementación simple sin coherencia cache: mantener las variables cerrojo en memoria.
  - o Sup la variable cerrojo está en  $M[0+(R1)]$

```
                DADDUI        R2,R0,#1
lockit:  EXCH        R2,0(R1)    ;atomic exchange
                BNEZ        R2,lockit    ;already locked?
```

- o Los accesos a mem implican tráfico en el sistema de comunicaciones → cuello de botella cuando varios procesadores compiten por el cerrojo
- o Si tuviéramos un sistema con coherencia cache, la situación no sería mucho mejor. Varios procesadores ejecutando EXCH → fallo de escritura → invalidaciones. ¿Cómo mejorar?

# Implementación de spin locks

- ❑ Implmentación con coherencia cache.
  - o Hacer una consulta preliminar del cerrojo, antes de hacer el exchange atómico. (test...exchange)
  - o Un procesador permanece leyendo su copia local mientras detecta que el cerrojo está bloqueado.
  - o Cuando lee en su cache que el cerrojo está libre (cómo se enteran?), ejecuta un EXCH y compite con los demás procesadores que estén haciendo el mismo spin lock. Uno de ellos consigue el bloqueo del cerrojo.
  - o El ganador verá un 0 en el registro de la instr EXCH, los demás verán un 1.
  - o Evita invalidaciones inútiles cuando un procesador está ejecutando repetidamente el bucle de espera.

```
lockit:    LD      R2,0(R1)    ;preliminary load of lock
           BNEZ R2,lockit    ;test 1: if not available → spin
           DADDUI   R2,R0,#1    ;R2 = 1
           EXCH R2,0(R1)    ;M[0+(R1)] ↔ R2
           BNEZ R2,lockit    ;test 2: if lock wasn't 0 → spin
```

- o Cuando el procesador ganador libera el cerrojo (escribiendo un 0) la carrera entre los perdedores empieza de nuevo.

# Implementación de spin locks

## ❑ Ejemplo: uso del código precedente en 3 procesadores

Paso	P0	P1	P2	Estado coherencia	Bus/actividad directorio
1	Ha adquirido el lock (lock = 1)	No copia de lock en cache. Lectura lock → fallo. Se sirve el fallo y pasa a compartido. Chequea si lock = 0 ( <b>test1</b> )	No copia de lock en cache. Lectura lock → fallo. Se sirve el fallo y pasa a compartido. Chequea si lock = 0 ( <b>test1</b> )	P0: M P1: I P2: I	Fallos de cache en P1 y P2. Estado lock pasa a shared
2	Libera lock (lock=0). Acierito de escritura. Invalida P1 y P2	Recibe invalidación	Recibe invalidación	P0: M P1: I P2: I	BusRdx de P0 para pasar a M. P1 y P2 ven BusRdX y pasan a invalidado
3		Fallo de cache (BusRd)	Fallo de cache (BusRd)	P0: S P1: I P2: S	Bus sirve primero fallo en P2 (flush de P0) y pasa a compartido en P0 y P2
4		Espera que se libere el bus para poder satisfacer su fallo	Chequea si lock=0 ( <b>test1</b> ), siendo el test positivo	P0: S P1: I P2: S	Fallo en P2 satisfecho, en P1 todavía no
5		Chequea si lock=0 ( <b>test1</b> ), siendo el test positivo	Como test1 positivo, ejecuta exchange	P0: S P1: S P2: S	Se satisface el fallo de P1
6		Como test1 positivo, ejecuta exchange	Resultado Exchange: 0 ( <b>éxito en test2</b> ) y pone lock =1 (adquiere el cerrojo)	P0: I P1: I P2: M	P2 escribe sobre bloque compartido (BusRdX). Lock exclusivo de P2
7		Resultado Exchange: 1 ( <b>fracaso en test2</b> ) y pone lock =1 ( NO adquiere el cerrojo)	Entra en la sección crítica	P0: I P1: M P2: I	Se sirve fallo cache P1, que envía invalidación (BusRdx) ya que escribe el lock. Flush de P2
8		Vuelve a testear si lock=0 ( <b>test1</b> )			Nada

# Implementación de spin locks

- ❑ Implementación usando LL+SC (evitar el uso de lectura-modificación-escritura atómica)

```
lockit:  LL          R2,0(R1)    ;load linked of lock (1)
        BNEZ R2,lockit    ;not available-spin (2)
        DADDUI   R2,R0,#1 ;R2=1
        SC          R2,0(R1)    ;store conditional
        BEQZ R2,lockit    ;branch if store fails (3)
```

- (1) El LL no tiene por qué generar tráfico en el sistema de comunicaciones
- (2) Si  $R2 \neq 0 \rightarrow$  el cerrojo está ocupado
- (3) Si  $R2 = 0 \rightarrow$  el cerrojo estaba libre cuando se ejecutó el LL, pero el SC ha fallado. Ha habido una interrupción, ha habido otra escritura (invalidación)... entre LL y SC  $\rightarrow$  Seguir intentando.

- ❑ ¿Qué aporta la coherencia?
  - o Las escrituras sobre una posición se hacen visibles a los lectores en el mismo orden
  - o Pero... no dice nada sobre el momento en que una escritura particular se hará visible para un lector concreto
- ❑ En un programa paralelo interesa frecuentemente que una lectura devuelva el resultado de una escritura particular
- ❑ **Ejemplo:** Consideramos 2 procesos, P1 y P2, ejecutándose en diferentes procesadores. P2 debe mostrar el valor de una variable compartida, A, después de que haya sido actualizada por P1.
  - o Para implementar este comportamiento, se suele recurrir a soluciones de código en las que suelen estar involucrados **más de una posición de memoria** (i.e., variable) **compartida entre P1 y P2** (leída y/o escrita por P1 y P2).

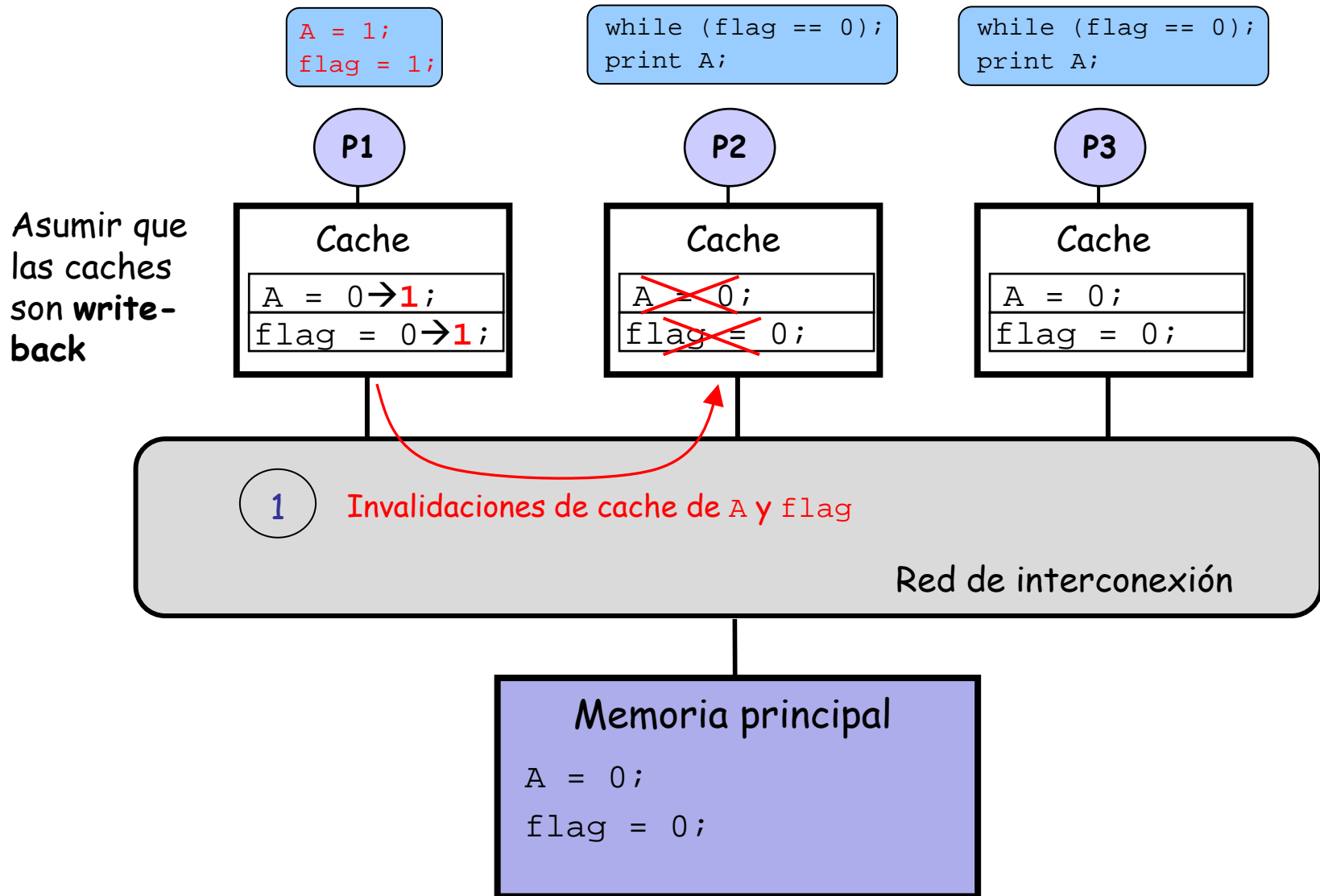
# Consistencia de memoria: Ejemplo 1

## ❑ Ejemplo 1: Sincronización mediante un flag.

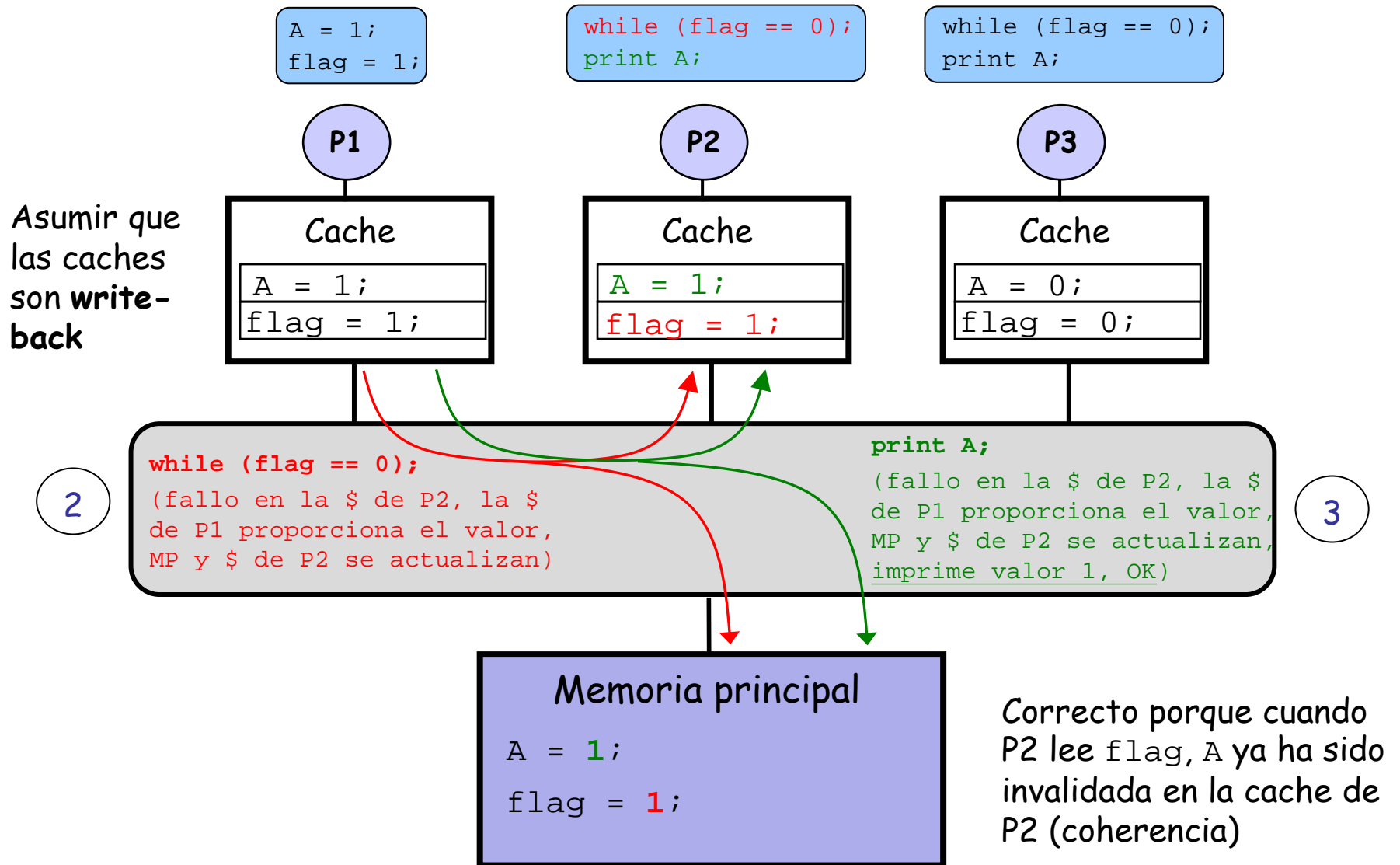
P1	P2
/*Valor inicial de A y de flag = 0*/	
A = 1;	while (flag == 0); /*espera activa*/
flag = 1;	print A;

- ❑ P1 almacena un nuevo valor de A, P2 debe leer el nuevo valor e imprimirlo.
  - o En teoría la variable `flag` forzaría este comportamiento.
- ❑ El programador está asumiendo que el resultado de `print A` debe ser 1, es decir:
  - o La escritura de la variable A se hace visible al P2 antes que la escritura de `flag`.
  - o La lectura de `flag` en el `while (flag == 0);` se ha completado antes de la lectura de A en el `print A`.
  - o Pero... Esta ordenación entre los accesos de P1 y P2 no está garantizada por la coherencia.
- ❑ La coherencia exige que el nuevo valor de A se haga visible a P2, pero no necesariamente antes que se haga visible el nuevo valor de `flag`.
  - o Luego, la coherencia por sí sola no evita que el resultado de `print A` sea 0.
- ❑ En general, la coherencia no dice nada sobre el orden con el que se hacen visibles las escrituras a diferentes posiciones de memoria ...
- ❑ Esperamos más de un sistema de memoria que simplemente "devolver el último valor escrito" para cada posición. Ese "último valor escrito" por P1 podría no ser el valor que se espera obtener en una lectura hecha por otro procesador P2.

# Consistencia de memoria: Ejemplo 1



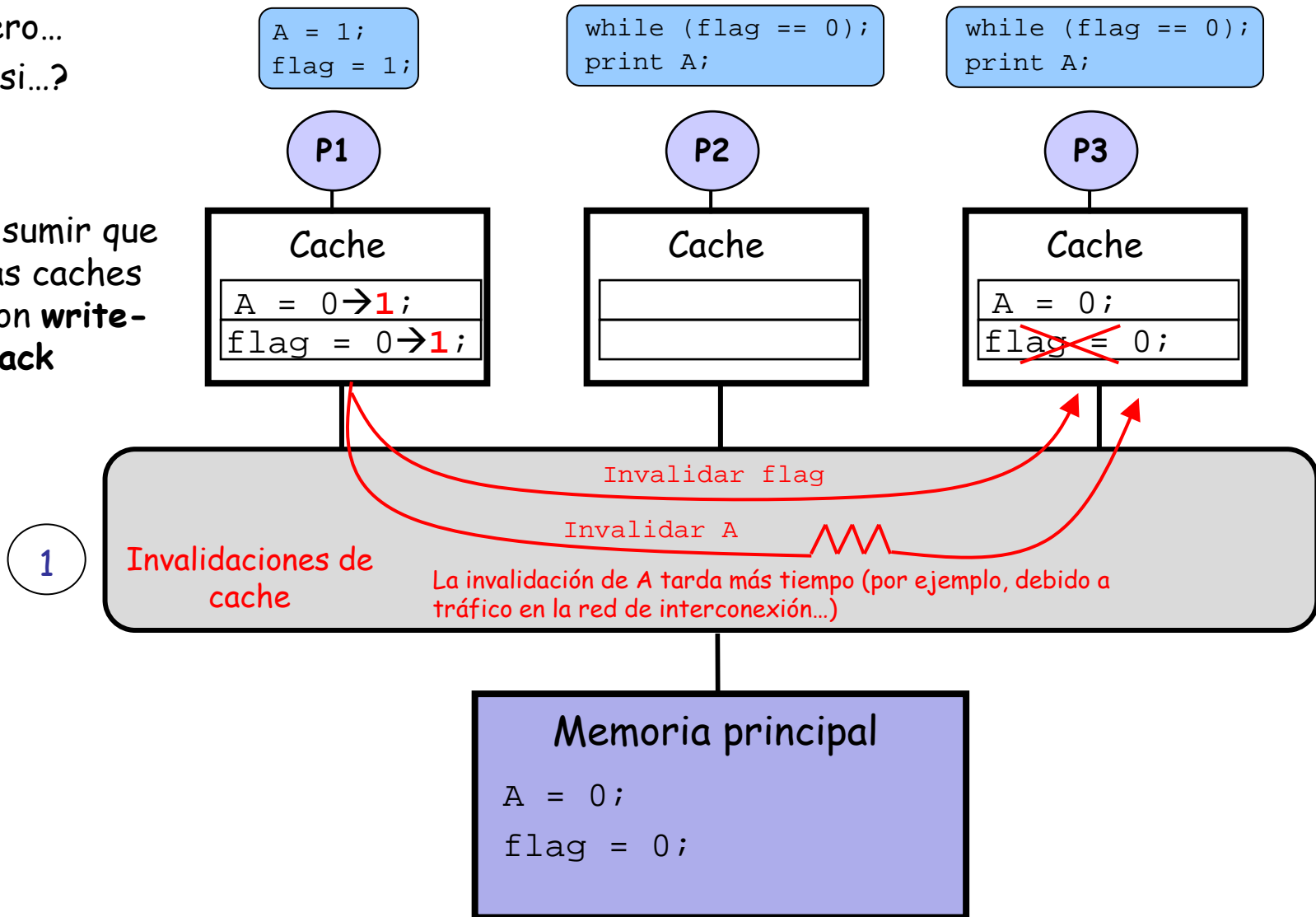
# Consistencia de memoria: Ejemplo 1



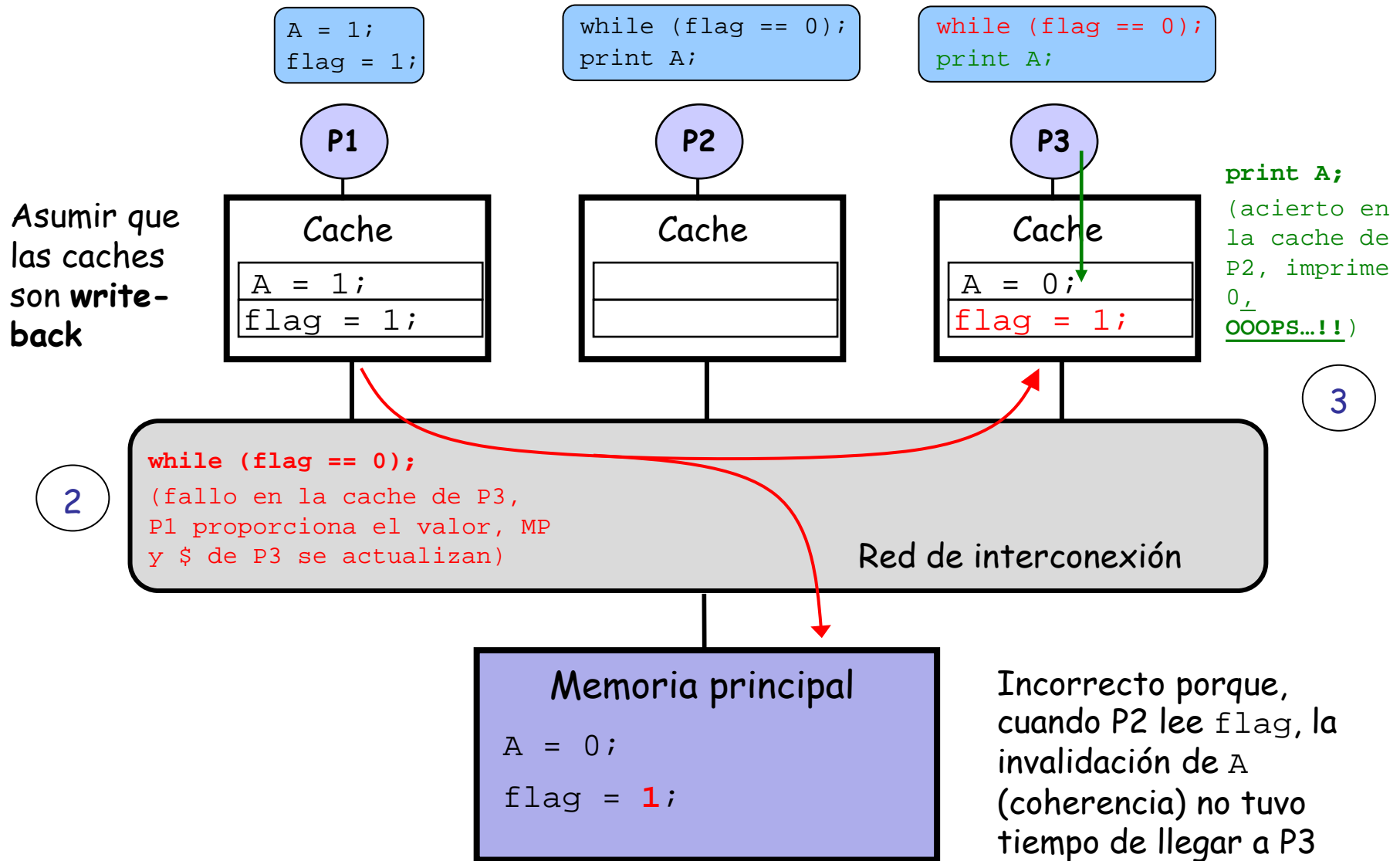
# Consistencia de memoria: Ejemplo 1

Pero...  
Y si...?

Asumir que  
las caches  
son **write-back**



# Consistencia de memoria: Ejemplo 1



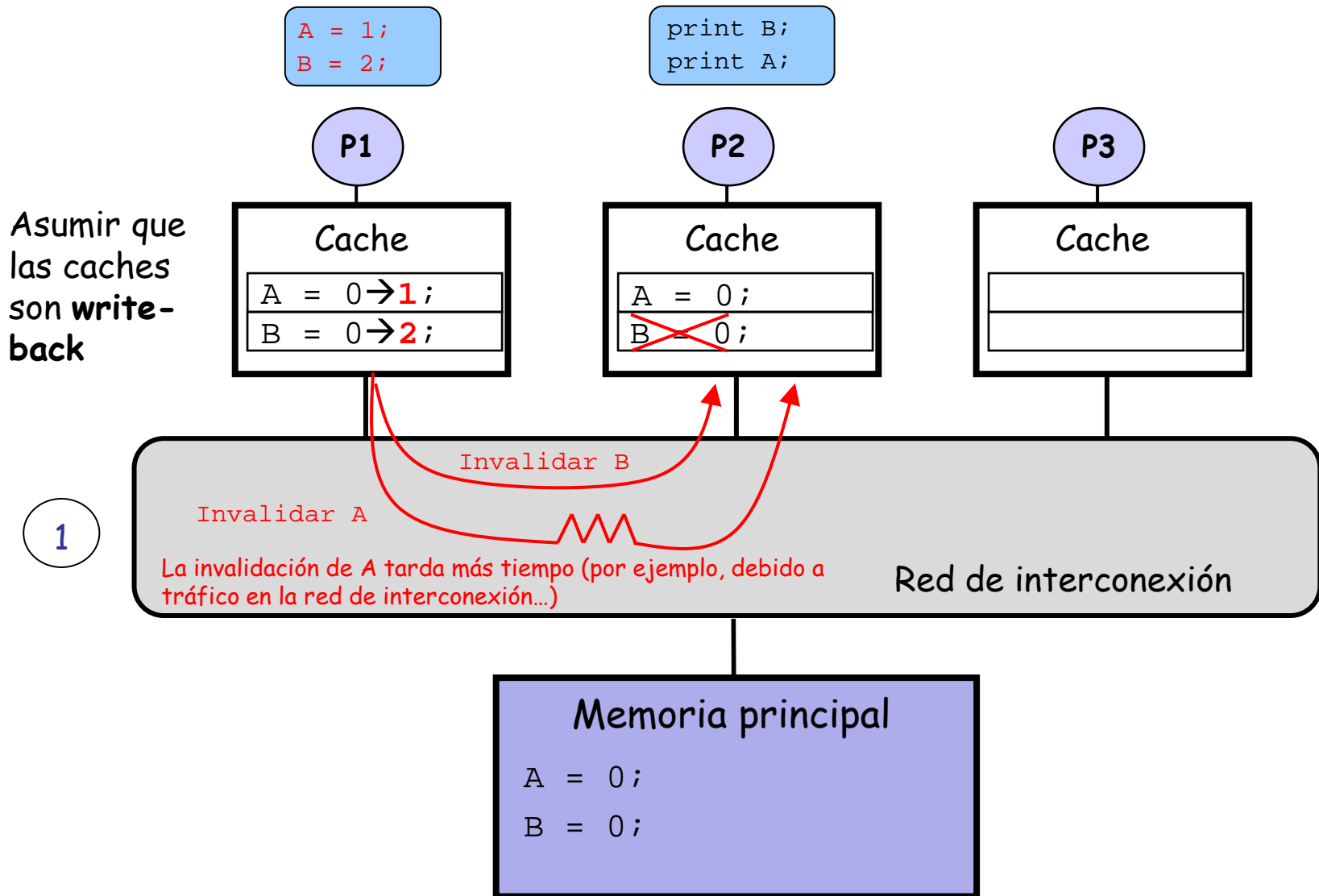
# Consistencia de memoria: Ejemplo 2

- ❑ No es preciso que exista comunicación explícita de procesos para tener comportamientos que resulten extraños desde el punto de vista del programador.
- ❑ **Ejemplo 2:** Procesadores no sincronizados, pero variables compartidas.

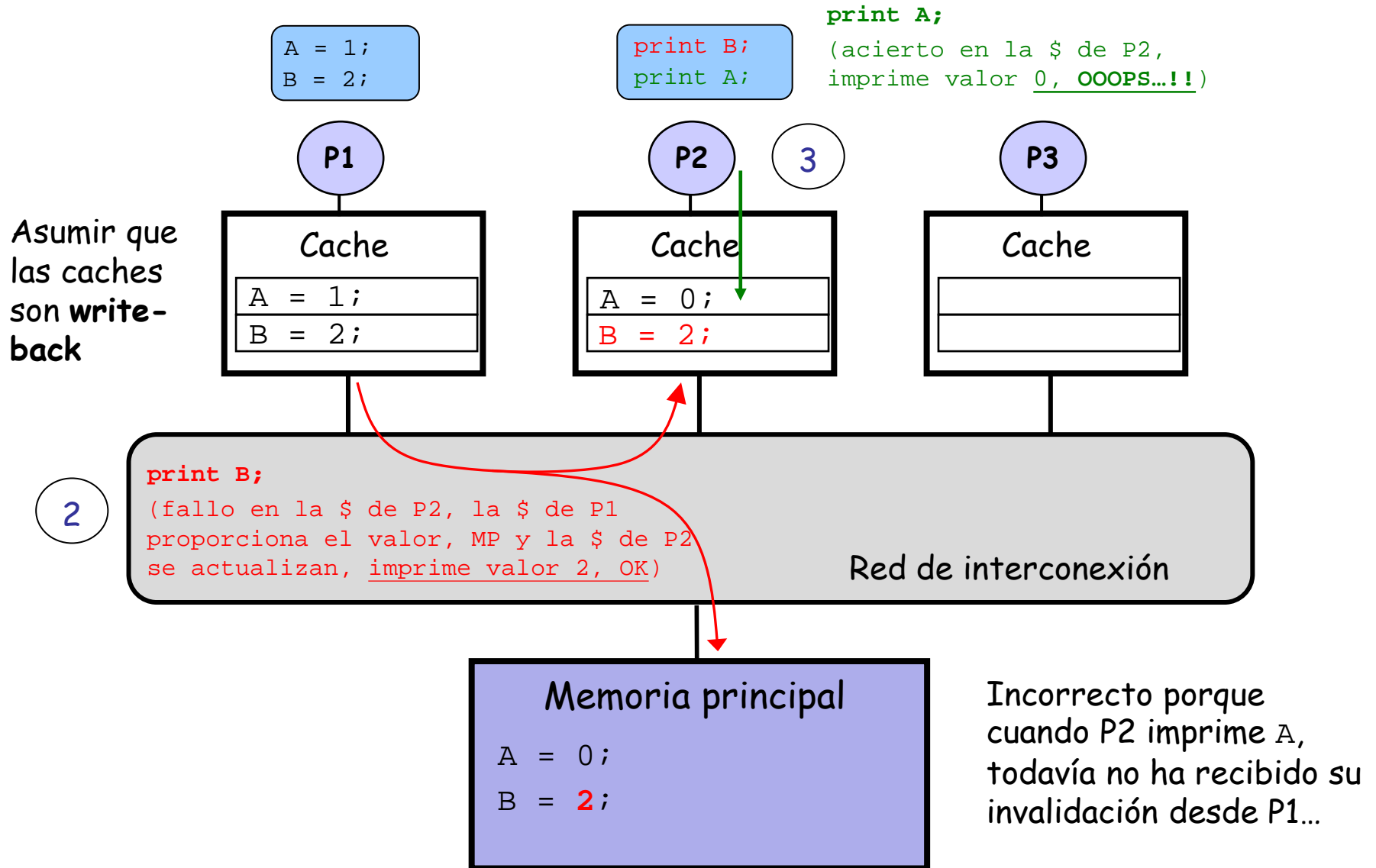
P1	P2
/* Valor inicial de A y B = 0 */	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

- o Los accesos a memoria hechos por los procesadores P1 y P2 son simples escrituras.
  - o A y B no se usan como variables de sincronización, pero **siguen siendo variables compartidas entre P1 y P2**.
    - La intuición nos dice que, si `print B` imprime un 2, entonces `print A` debería imprimir un 1.
  - o Pero... 2a y 2b leen diferentes direcciones de memoria, modificadas por P1. Y la coherencia no dice nada sobre el orden en que esas escrituras se hacen visibles a P2.
  - o Por tanto, si P1 ejecuta (1a)→(1b), **sin asegurarse antes de que las invalidaciones de A se han completado al 100% para mantener la coherencia**, podría ocurrir que el resultado de `print B` fuese 2 y el de `print A` fuese 0. Es decir, P2 leería A de su cache local (aún 0) porque esa invalidación proveniente de P1 (provocada por (1a) A=1) aún no le ha llegado.
  - o Además, optimizaciones del compilador y/o ejecución fuera de orden en el procesador puede alterar el orden de ejecución de (1a)→(1b) o y/o (2a)→(2b).
- ❑ Por tanto, el **orden de programa** (p. ej., 1a <<sub>p</sub> 1b) y el **orden de memoria** (1a <<sub>m</sub> 1b) de los loads y stores no son necesariamente siempre el mismo.

# Consistencia de memoria: Ejemplo 2



# Consistencia de memoria: Ejemplo 2



# Consistencia de memoria

- ❑ Necesitamos algo más que la coherencia para dar a un espacio de direcciones compartido una semántica clara.
- ❑ Es necesario un modelo de ordenación que los programadores puedan usar para razonar acerca de la corrección de los programas.
- ❑ **Modelo de consistencia de la memoria para un espacio de direcciones compartido:** especifica las restricciones en el orden en el que las operaciones de memoria (emitidas por un único o por diferentes procesos) deben hacerse visibles a los procesadores.
  - o Incluye operaciones a las mismas o a diferentes posiciones de memoria → La consistencia incluye a la coherencia.
- ❑ **Contrato entre el programador y el diseñador del sistema**
  - o Si el programador sigue las reglas establecidas por el HW para las operaciones en memoria, ésta será consistente.
  - o Para el compilador, el modelo impone ciertas limitaciones a las optimizaciones.
  - o Para el programador, el modelo proporciona información sobre qué operaciones de memoria pueden finalizar fuera de orden respecto a cuáles.

# Consistencia Secuencial (CS)

## ❑ El modelo de consistencia más restrictivo es el de Consistencia Secuencial (CS):

- o Este modelo impone que el orden en el que las operaciones de memoria (loads y stores) se realizan en el código (**orden de programa**) sea el mismo que en el que estas operaciones se completan en memoria (**orden de memoria**).
- o Por tanto, obliga a:  $L \rightarrow S$ ,  $L \rightarrow S$ ,  $S \rightarrow S$ ,  $L \rightarrow L$ .
- o En CS, el **orden de memoria** siempre respeta el **orden de programa**, lo cual no es cierto en otros modelos de consistencia.

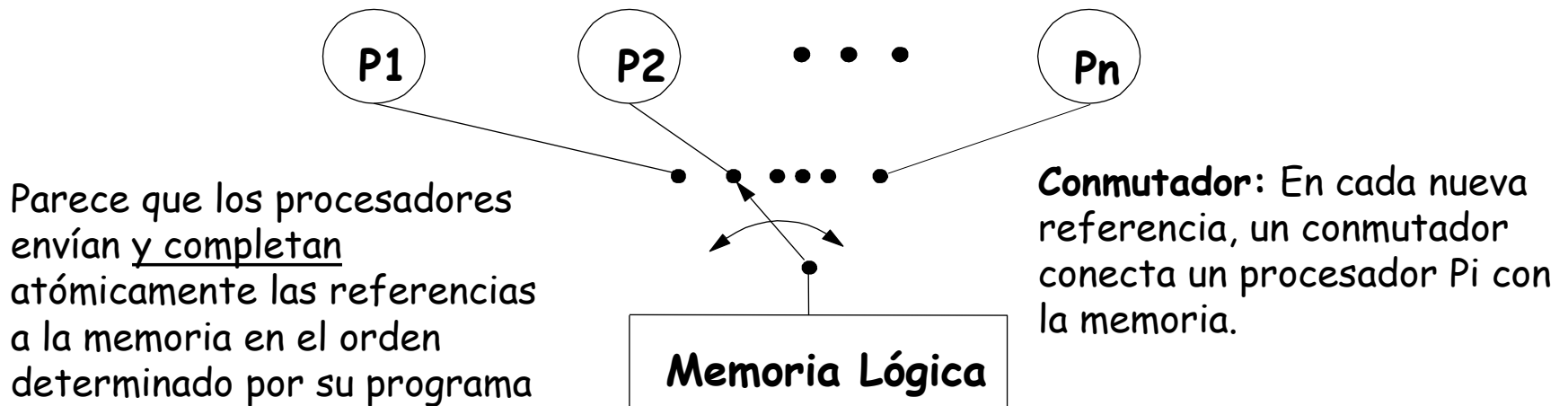
## ❑ Formalismo:

- o Orden de programa:  $\langle p$
- o Orden de memoria:  $\langle m$
- o En el caso de CS:
  - Si  $L(a) \langle p L(b) \rightarrow L(a) \langle m L(b)$  //Load  $\rightarrow$  Load (o  $L \rightarrow L$ )
  - Si  $L(a) \langle p S(b) \rightarrow L(a) \langle m S(b)$  //Load  $\rightarrow$  Store (o  $L \rightarrow S$ )
  - Si  $S(a) \langle p S(b) \rightarrow S(a) \langle m S(b)$  //Store  $\rightarrow$  Store (o  $S \rightarrow S$ )
  - Si  $S(a) \langle p L(b) \rightarrow S(a) \langle m L(b)$  //Store  $\rightarrow$  Load (o  $S \rightarrow L$ )
  - Cada load  $L(a)$  obtiene el valor del "último" store realizado antes que  $L$  en la posición de memoria  $a$ , siendo el "último" store en orden global en memoria (teniendo en cuenta todos los cores).

# Consistencia Secuencial (CS)

## ❑ ¿Cómo ve el programador la memoria si se utiliza CS?

- o Desde fuera, pareciera que los procesadores comparten una única memoria (aunque realmente ésta se halle distribuida ente los procesadores, cada uno con su cache).



# Consistencia Secuencial (CS)

## ❑ Ejemplo

P1	P2
/*Valor inicial de A y B = 0*/	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

## ❑ Resultados posibles bajo CS:

- o Orden 2a, 2b, 1a, 1b produce  $(A,B) = (0,0)$
- o Orden 1a, 2a, 1b, 2b produce  $(A,B) = (1,0)$
- o Orden 1a, 1b, 2a, 2b produce  $(A,B) = (1,2)$

## ❑ Es posible el resultado $(A,B) = (0,2)$ bajo CS?

- o El orden del programa implica  $1a \rightarrow 1b$  y  $2a \rightarrow 2b$
- o  $A = 0$  implica  $2b \rightarrow 1a$ , lo que implica  $2a \rightarrow 1b$
- o  $B = 2$  implica  $1b \rightarrow 2a$ , **contradicción**

# ¿Es siempre conveniente la CS?

- ❑ Supongamos el siguiente ejemplo:

P1	P2
(1a) <code>i = 0;</code>	(2a) <code>j = 127;</code>
(1b) <code>A = X[i];</code>	(2b) <code>X = Y[j];</code>
(1c) <code>if (A == 2) then B = 0;</code>	(2c) <code>if (X == 4) then Y = 0;</code>

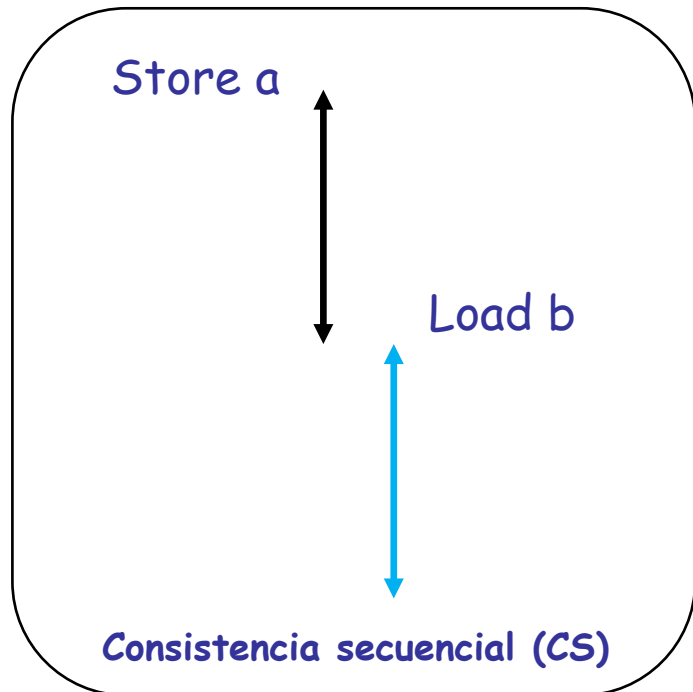
- ❑ P1 y P2 ejecutan en paralelo códigos totalmente independientes entre sí.
- ❑ **Problema:** Si se fuerza la CS para "respetar el orden de programa", cada nuevo acceso a memoria no puede realizarse hasta que el anterior no finalice por completo.
- ❑ En este ejemplo, no es necesario que ninguna de las instrucciones de P1 (1a, 1b, 1c) o P2 (2a, 2b, 2c) espere a que finalice el acceso a memoria actualmente en curso por el otro procesador (en caso de que éste se esté produciendo) para poder comenzar.
- ❑ Y en general, tampoco es necesario **que el hardware respete en memoria (<m) el orden de programa (<p).**
  - o Se está incurriendo en **pérdidas de rendimiento** innecesarias:
    1. No permite solapar lecturas y escrituras independientes entre sí (HW).
    2. Limita enormemente las optimizaciones de compilador (SW).

# Total Store Ordering (TSO)

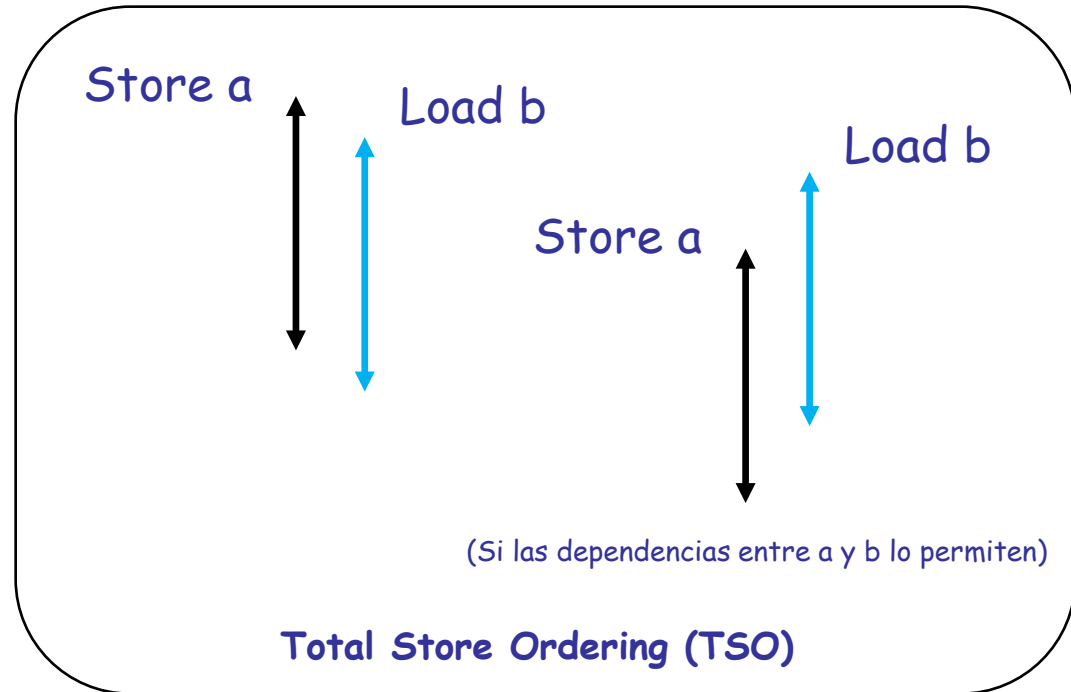
❑ **Optimización:** Permitir reordenar operaciones de  $S \rightarrow L$ .

❑ **Definición formal**

- Si  $L(a) \prec_p L(b) \rightarrow L(a) \prec_m L(b) // \text{Load} \rightarrow \text{Load (o } L \rightarrow L)$
- Si  $L(a) \prec_p S(b) \rightarrow L(a) \prec_m S(b) // \text{Load} \rightarrow \text{Store (o } L \rightarrow S)$
- Si  $S(a) \prec_p S(b) \rightarrow S(a) \prec_m S(b) // \text{Store} \rightarrow \text{Store (o } S \rightarrow S)$
- ~~Si  $S(a) \prec_p L(b) \rightarrow S(a) \prec_m L(b) // \text{Store} \rightarrow \text{Load (o } S \rightarrow L)$~~



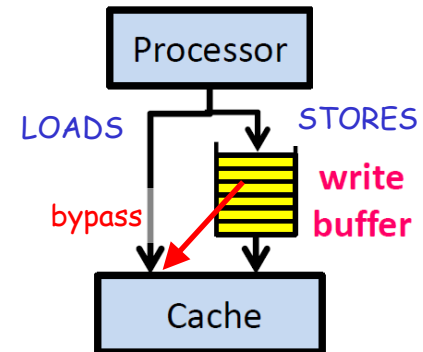
vs.



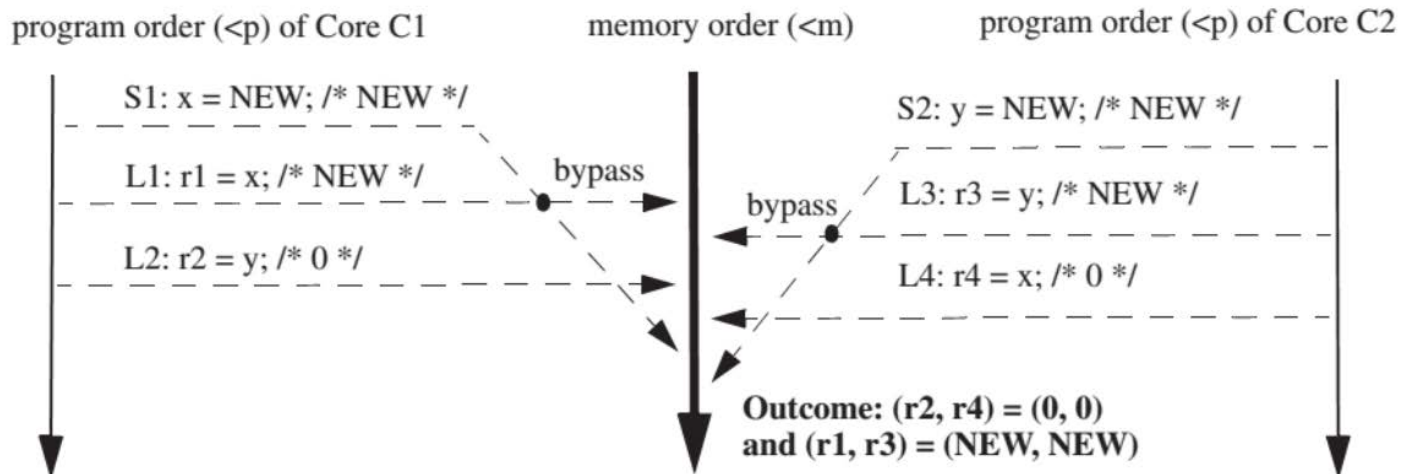
# Total Store Ordering (TSO)

## ❑ Solución de implementación típica en TSO: **Write buffer**

- o Los stores finalizan en orden entre sí (preserva  $S \rightarrow S$ ).
- o Los loads pueden adelantar a los stores si las dependencias lo permiten (relaja  $S \rightarrow L$ ).
- o Misma idea que tema 3 ("Store queue").



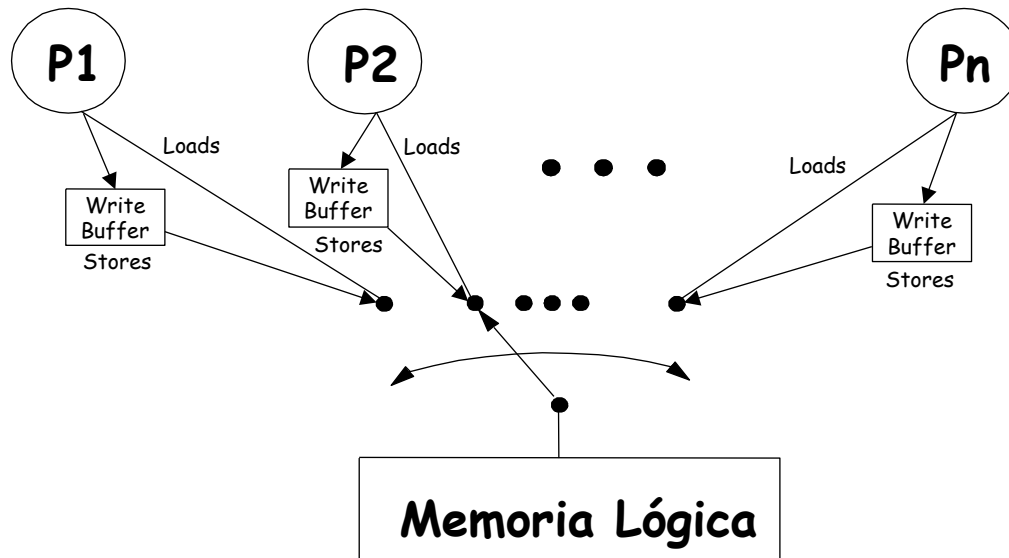
- ## ❑ Para que un load pueda adelantar "de manera segura" a un store dependiente, anterior y ya almacenado en el buffer (contexto de un único core), es necesaria una **red de bypass**:



# Total Store Ordering (TSO)

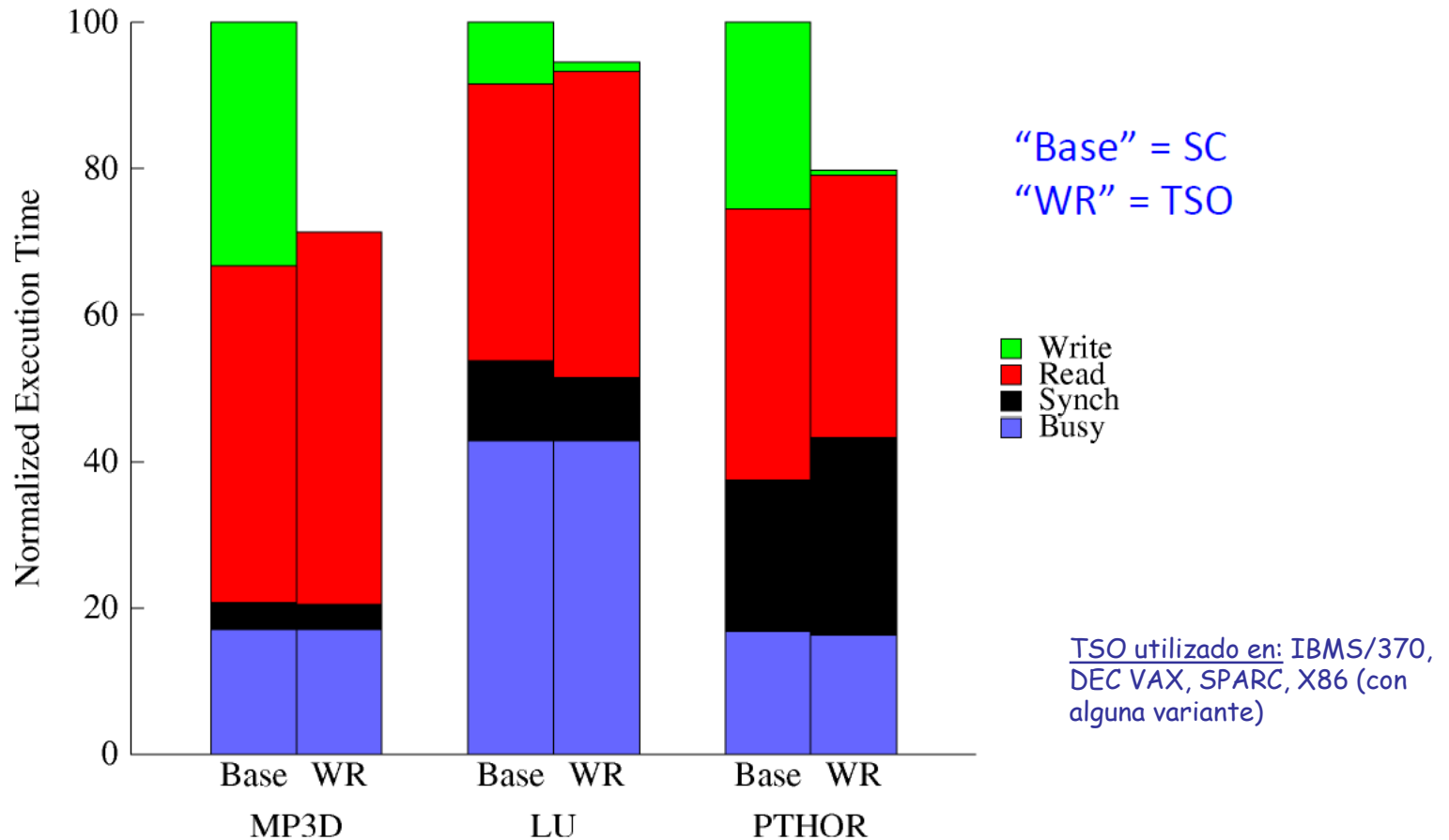
## ❑ ¿Cómo ve el programador la memoria si se utiliza TSO?

- o Un sistema muy similar al que se tiene con CS, pero esta vez cada core tiene un buffer FIFO para mantener los stores en orden
- o El buffer incluye lógica de bypass para permitir que los loads adelanten "de manera segura" a stores anteriores de los que dependen (contexto de un único core).



# Total Store Ordering (TSO)

## □ Rendimiento CS vs. TSO:



[http://www.cs.cmu.edu/afs/cs/academic/class/15418-s18/www/lectures/13\\_consistency.pdf](http://www.cs.cmu.edu/afs/cs/academic/class/15418-s18/www/lectures/13_consistency.pdf)

# Partial Store Ordering (PSO)

- ❑ **Optimización:** Permitir reordenar operaciones de  $S \rightarrow S$ .
- ❑ **Definición formal**
  - Si  $L(a) \prec_p L(b) \rightarrow L(a) \prec_m L(b) // \text{Load} \rightarrow \text{Load} \text{ (o } L \rightarrow L)$
  - Si  $L(a) \prec_p S(b) \rightarrow L(a) \prec_m S(b) // \text{Load} \rightarrow \text{Store} \text{ (o } L \rightarrow S)$
  - ~~Si  $S(a) \prec_p S(b) \rightarrow S(a) \prec_m S(b) // \text{Store} \rightarrow \text{Store} \text{ (o } S \rightarrow S)$~~
  - ~~Si  $S(a) \prec_p L(b) \rightarrow S(a) \prec_m L(b) // \text{Store} \rightarrow \text{Load} \text{ (o } S \rightarrow L)$~~
- ❑ Los stores ahora pueden adelantar a otros stores que sean independientes en un mismo core.
  - o Dependencias WAW en memoria, en el contexto de 1 core, son infrecuentes.
  - o Un segundo store puede adelantar a un store anterior que sea lento (p. ej., fallo de cache).
- ❑ Pero... esto vuelve a generar comportamientos "extraños" para el programador:

P1	P2
/* Valor inicial de A y de flag = 0 */	
A = 1;	while (flag == 0); /*espera activa*/
flag = 1;	print A;

# Weak Ordering (WO) y Release Consistency (RC)

❑ **Optimización:** Permitir reordenar cualesquiera operaciones

❑ **Definición formal**

- ~~$Si\ L(a) \leftarrow p\ L(b) \rightarrow L(a) \leftarrow m\ L(b) // Load \rightarrow Load\ (o\ L \rightarrow L)$~~
- ~~$Si\ L(a) \leftarrow p\ S(b) \rightarrow L(a) \leftarrow m\ S(b) // Load \rightarrow Store\ (o\ L \rightarrow S)$~~
- ~~$Si\ S(a) \leftarrow p\ S(b) \rightarrow S(a) \leftarrow m\ S(b) // Store \rightarrow Store\ (o\ S \rightarrow S)$~~
- ~~$Si\ S(a) \leftarrow p\ L(b) \rightarrow S(a) \leftarrow m\ L(b) // Store \rightarrow Load\ (o\ S \rightarrow L)$~~

❑ Desde el punto de vista de un único core, es la opción más eficiente posible.

❑ Para varios cores: Los comportamientos "extraños" se vuelven mucho más frecuentes si se utilizan variables compartidas para sincronizar hilos de ejecución.

❑ **Solución:** Utilizar primitivas de sincronización.

o Ejemplo Intel X86 (TSO)

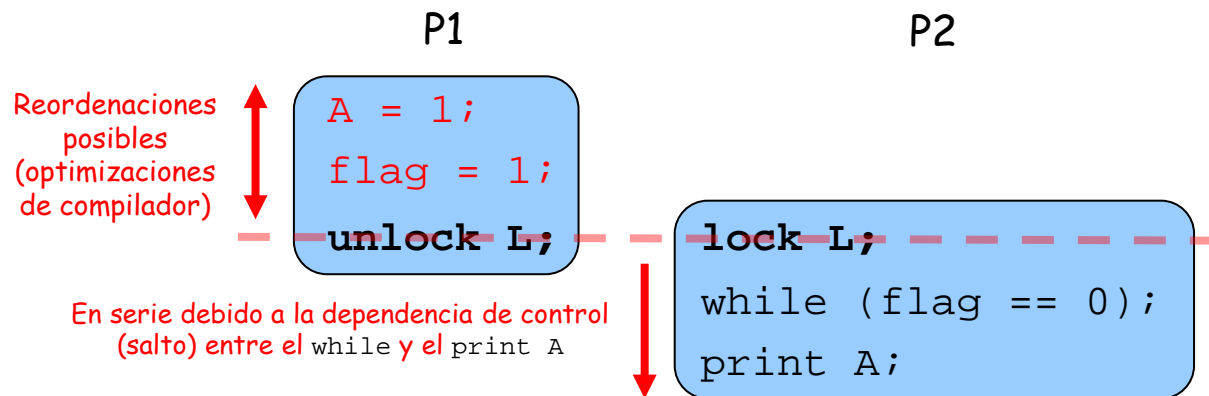
- `mm_1fence` ("load fence": esperar a que se completen todos los LDs anteriores).
- `mm_sfence` ("store fence": esperar a que se completen todos los STs anteriores).
- `mm_mfence` ("mem fence": esperar a que se completen todas las operaciones de memoria).

o Otras primitivas útiles (dependen de la ISA): `lock(L)` – `unlock(L)`

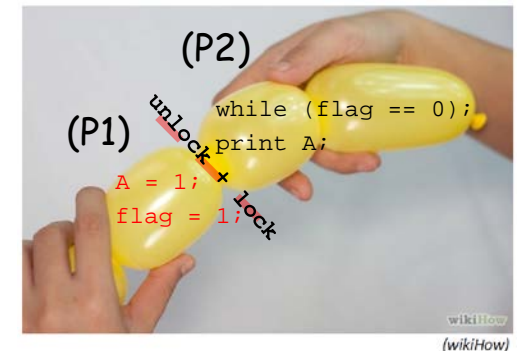
- Idea de "adquirir" y "soltar" un cerrojo para garantizar la exclusión mutua entre cores.

# Primitivas de sincronización + modelos de consistencia relajados

- ❑ **Idea:** La mayoría de los programas sólo necesitan que el orden de memoria respete al orden de programa en ciertas ocasiones.
  - o Cuando existen accesos conflictivos: realizados por cores distintos (P1 y P2) a la misma variable compartida y, **al menos, uno de ellos, es una escritura.**
    - `A = 1;` + `print A;`
    - `flag = 1;` + `while (flag == 0);`
  - o Se puede forzar a que estos accesos respeten el orden de programa introduciendo primitivas de sincronización. Por ejemplo:



Forzamos a que el código de P1 finalice ("commit") antes que el código de P2

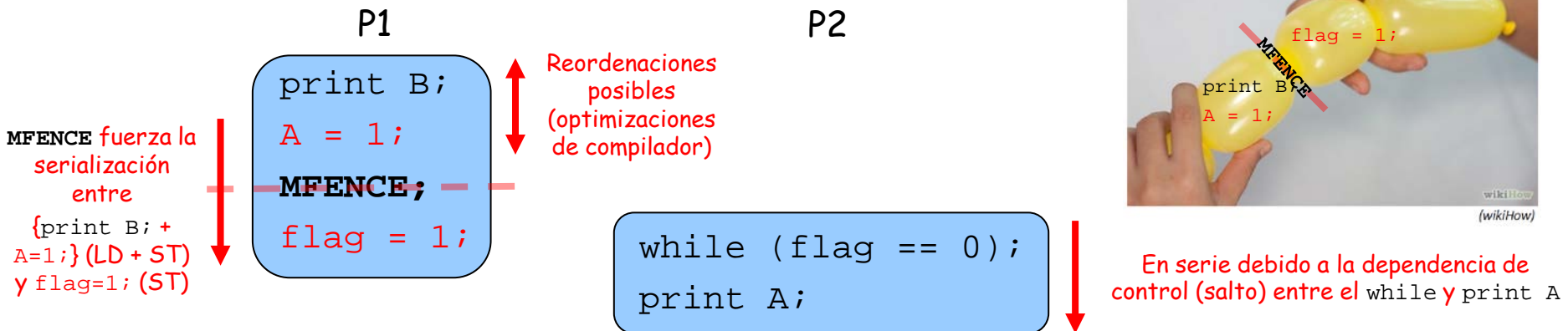


- ❑ **Estas primitivas se deben utilizar para eliminar las "data races":** Accesos conflictivos a la misma variable compartida que hace que el programa tenga un comportamiento no determinista.
  - o Su resultado depende aleatoriamente de la velocidad relativa entre los procesadores involucrados.

# Primitivas de sincronización + modelos de consistencia relajados

## ❑ Si la ISA proporciona primitivas **MFENCE**:

- o Esta operación se puede utilizar para serializar 2 accesos consecutivos a memoria.
- o En el HW: Implica hacer esperar al procesador a que finalicen "commit" todos los LDs y STs previos antes de continuar.



## ❑ **MFENCE** es una operación muy restrictiva y costosa

- o Requiere esperar a que todas las operaciones de memoria anteriores hayan hecho commit.
  - Quizá no sea necesario ser tan restrictivo (por ejemplo, no sería incorrecto que `print B;` se ejecute después de `flag=1;`)

# Primitivas de sincronización + modelos de consistencia relajados

## ❑ Si la ISA proporciona primitivas **LFENCE** y **SFENCE**:

- **LFENCE**: Se pueden serializar sólo los LD a ambos lados del **LFENCE** (los ST pueden atravesarlo).
- **SFENCE**: Se pueden serializar sólo los ST a ambos lados del **SFENCE** (los LD pueden atravesarlo).

P1

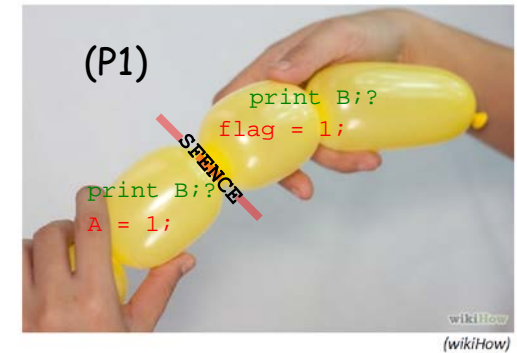
```
print B;  
A = 1;  
SFENCE;  
flag = 1;
```

**SFENCE** sólo  
fuerza la  
serialización  
entre A=1; (ST)  
y flag=1; (ST)

P2

```
while (flag == 0);  
print A;
```

print B;  
puede atravesar  
**SFENCE** pero  
A=1; no puede



En serie debido a la dependencia de  
control (salto) entre el while y print A

- ## ❑ **LFENCE** y **SFENCE** permiten más optimizaciones de compilador pero requiere mayor esfuerzo por parte del programador y/o compilador.

# Resumen de la consistencia

- ❑ En un sistema multiprocesador, el uso de variables compartidas entre cores puede generar comportamientos “extraños” o “inesperados” para el programador.
  - o Esto es debido a las **optimizaciones de compilador** y a que las **operaciones de memoria no finalizan** (desde el punto de vista de la coherencia) **en el mismo orden en el que se lanzaron** (por ejemplo, debido a diferentes retardos en la red de interconexión).
- ❑ La **Consistencia Secuencial (CS)** garantiza que el **orden de memoria (<m)** de los loads y stores sea el mismo que su **orden de programa (<p)**.
  - o Problema: Ineficiente y no siempre conveniente ser tan restrictivo.
- ❑ Otros **modelos de consistencia** establecen un “contrato” entre el programador y el HW sobre qué reordenaciones de compilador son posibles.
  - o Permiten **relajar la CS** para permitir ciertas reordenaciones de operaciones de memoria (loads y stores).
  - o Uso de **directivas de sincronización**: `lock(L) + unlock(L)`, `MFENCE`, `LFENCE`, `SFENCE`... para forzar serialización entre operaciones de memoria.
  - o Más **eficientes** pero más **esfuerzo** de programador y compilador.