Tema 3: Multiprocesadores
Parte 2: Sincronización
Consistencia

Universidad de Cádiz

Sincronización de procesos en SMP

La ejecución entre procesos de distintos procesadores no es independiente…

… necesitan pasarse datos mediante variables compartidas …

y necesitan **….Sincronizar su ejecución**

Proc. 1

```
.....
Dato1=50
.....
Dato1=15
.....
```

Proc. 2

```
.....
.....
Print Dato1
.....
.....
```

¿Qué imprimiría el procesador 2?

Sincronización de procesos en SMP

Al usar variables compartidas hay que tener cuidado con las dependencias (o antidependencias)

Hay que asegurar la lógica del programa con mecanismos de sincronización

Proc. 1

```
.....
LD    r1, Dat0
AddI  r1,r1,1
ST    r1, Dat0
.....
```

Proc. 2

```
.....
LD    r1, Dat0
AddI  r1,r1,1
ST    r1, Dat0
.....
```

¿Qué Valor tendría Dat0 al acabar el proceso en cada procesador?

Este código tendría que ser **atómico** para funcionar correctamente

Sincronización de procesos en SMP

Sección Critica. Sección de código que para asegurar su correcto funcionamiento, en un momento dado solo puede ejecutarse en un procesador.

Cerrojo. Variables tipo semáforo a la entrada de una sección critica

Sincronización de procesos en SMP

Sincronización

- Por **hardware**. Incorporando instrucciones de sincronización

  Rápida

- Por **software**. Usando bibliotecas de sincronización

  Soluciones flexibles

Se suelen usar mezcla hardware/software

Sincronización de procesos en SMP

Estrategias de sincronización:

- **Exclusión mutua**

  Solo un proceso puede ejecutar una sección de código

- **Sincronización**

  Para ejecutar un proceso se espera un suceso o acción. Dos tipos:

  - Punto a punto (mediante *flags*)
  - Global (mediante barreras)

Sincronización de procesos en SMP

Algoritmos de espera:

- **Espera activa**
  Bucle continuo preguntando si ‘se puede pasar’ / ‘hacer una acción’…

- **Bloqueo**
  Se cambia de contexto, hasta que se pueda continuar/hacer. Gestionado por el SO

Sincronización de procesos en SMP

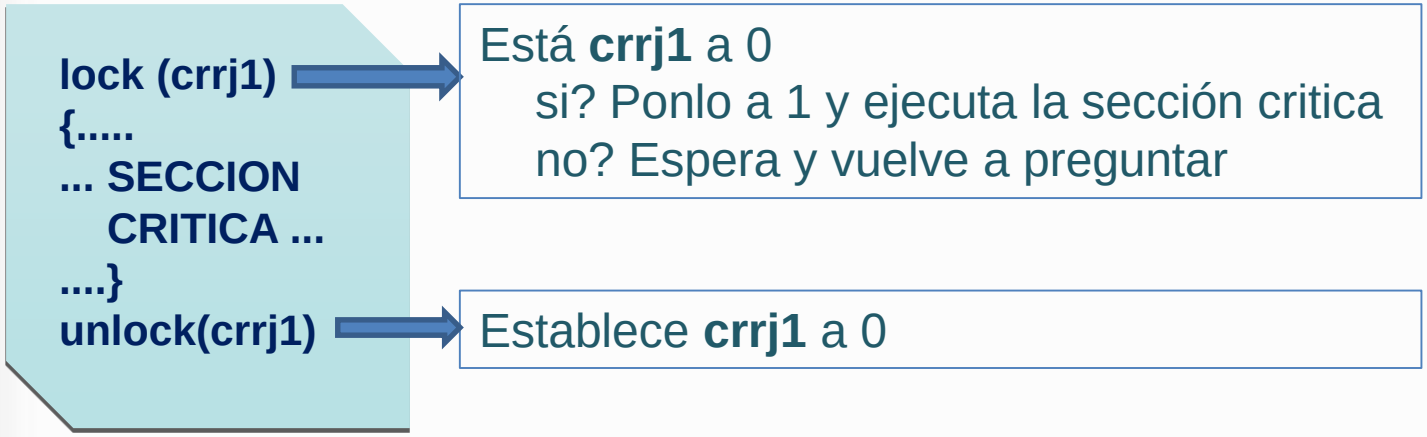
Características deseadas en una sincronización:

- Baja latencia
- Trafico limitado
- Escalabilidad
- Poco coste en memoria
- Igualdad de oportunidades

Exclusión Mutua

Se utiliza la *exclusión mutua* para evitar que una sección del código (**sección critica**) sea simultáneamente ejecutada por más de un proceso.

Usa funciones **Lock** y **Unlock** para cambiar y leer el valor del **cerrojo** asociado a esa *sección critica*



Puede implementarse por hardware insertando líneas de control pero lo habitual es implementarlo por software

Exclusión Mutua

```
lock (crrj1)
{.....
... SECCION
    CRITICA ...
....}
unlock(crrj1)
```

```
lock:
    LD $t1, crrj1
    BNZ $t1, look

    LI $t1,1
    ST $t1, crrj1
    RET
```

!!!

```
unlock:
    ST $zero, crrj1
    RET
```

Mismo problema de sincronismo con el cerrojo
2 procesos pueden acceder a la vez y generar conflictos.

Se necesitan instrucciones atómicas de tipo
RMW (Read Modify Write)

Todos los procesadores actuales tiene varias tipos de instrucciones RMW:

Exclusión mutua
Instrucciones RMW (Test&Set y Swap)

Test&Set

Primera instrucción sencilla y antigua.

De forma atómica lee el cerrojo y si es 0 lo pone a 1.

TS $t1, crrj1 → $t1 = mem[cerrj1]; mem[cerrj1] = 1;

```
lock:
    TS $t1, crrj1
    BNZ $t1, look
RET
```

```
unlock:
    ST $zero, crrj1
    RET
```

Swap

Intercambia de forma atómica el contenido de un registro con una posición de memoria

Exclusión mutua
Instrucciones RMW (Test&Set y Swap)

INCONVENIENTE

Pese a funcionar correctamente la instrucción Test & Set tiene el inconveniente de ser muy poco escalable y de degradar el sistema en situaciones de alta competencia.

Cada vez que se ejecuta la instrucción, escribe siempre aunque el cerrojo esté cerrado provocando una sobrecarga en el bus.

Exclusión mutua (Mejoras Test&Set)

Test&Set con espera

Si el cerrojo está cerrado espero un tiempo ‘prudencial’ antes de volver a preguntar si ya está libre.

El tiempo de espera no puede ser…

- …Ni muy alto.
  - Para que no pierda demasiado tiempo en esperar
- … Ni muy bajo.
  - Para que intentar entrar en vano.

Lo adecuado es usar un tiempo que se incremente exponencialmente con el numero de intentos fallidos en esa sección critica

Exclusión mutua (Mejoras Test&Set)

Test&Set con espera

```
lock:
    TS $t1, crrj1
    BNZ $t1, espera
RET

espera:
    WAIT t
    Calculo de próximo t
    JMP lock
```

```
unlock:
    ST $zero, crrj1
    RET
```

Exclusión mutua (Mejoras Test&Set)

Test&Test&Set

Se divide la operación en dos partes:

- Se lee normalmente el cerrojo hasta que esté abierto
- Se hace una instrucción TS

```
lock:
    LD $t1, crrj1
    BNZ $t1, espera

    TS $t1, crrj1
    BNZ $t1, espera
RET
```

```
unlock:
    ST $zero, crrj1
    RET
```

Exclusión mutua
Instrucciones RMW (LL y SC)

Otra forma de gestionar la exclusión mutua

LL – Load Locked

Lee de memoria y guarda en un registro especial que dirección ha leído y un flag de estado

SC – Store Condicional

Consulta el registro especial para ver si puede salvar el dato

```
LL $t1, var
      $t1 = MEM [var]
       LSin[dir] = var;    LSin[flag] = 1;
```

```
SC $t1, var
    si ( LSin[dir] = var y LSin[flag] = 1 )
          MEM[var] = $t1
          LSin[flag] = 0;
          $t1=1
    sino $t1=0
```

Exclusión mutua
Instrucciones RMW (LL y SC)

Ejemplo exclusión mutua con LL y SC

```
lock:
    LI $t2, 1

l1:
    LL $t1, crrj1
    BNZ  $t1, l1
        .....
        .....
    SC $t2, crrj1
    BZ  $t2, lock

RET
```

```
unlock:
    ST $zero, crrj1
    RET
```

Exclusión mutua
Instrucciones RMW (Compare&Swap)

CS $t1, $t2, crrj1 → Si ($t1 = mem[cerrj1]) mem[cerrj1] ←→ $t2;

CS efectúa el intercambio si se cumple la condición.
Esto se realiza con ayuda de hardware.

```
lock:
    LI $t2, 1

l1:
    CS $zero, $t2, crrj1
    BNZ $t2, l1

RET
```

```
unlock:
    ST $zero, crrj1
    RET
```

Exclusión mutua
Instrucciones RMW (Fetch&Op)

Las instrucciones RMW anteriores eran muy simples

Hay un grupo de instrucciones de uso más general y que antes de escribir en memoria realiza una operación.

Fetch&Incr $t1, var

$t1 = mem[var]; mem[var] = mem[var] + 1;

Fetch&Dcr $t1, var

$t1 = mem[var]; mem[var] = mem[var] - 1;

Fetch&Add $t1, $t2, var

$t1 = mem[var]; mem[var]=mem[var] + $t2;

Exclusión mutua
Métodos para reducir el trafico

Tickets

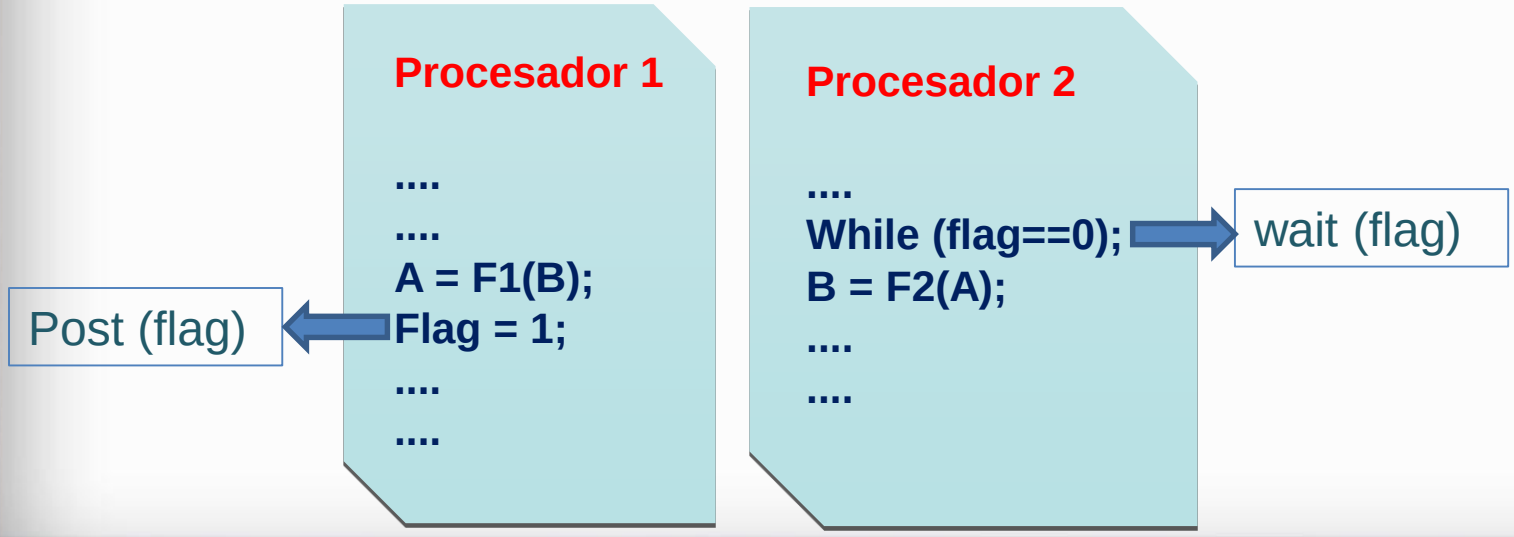
- Se implementa una cola (FIFO) para acceder a la sección critica. Los procesos esperaran su turno para acceder.
- Se implementa con dos variables *Ticket* y *Turno*.
- Las incrementos de estas variables se realizan con la instrucción *Fetch&Incr.* Además el tiempo de espera puede ser proporcional al *turno* obtenido.
- Se genera trafico al obtener el *ticket* y al actualizar el *turno*.

Sincronización Punto a Punto mediante eventos

La sincronización es punto a punto si solo participan 2 procesadores

Se realiza mediante un bucle de espera activa consultando una variable (flag) común.

Algunos procesadores incorporan instrucciones especificas **(post, wait)**



Sincronización mediante Barreras

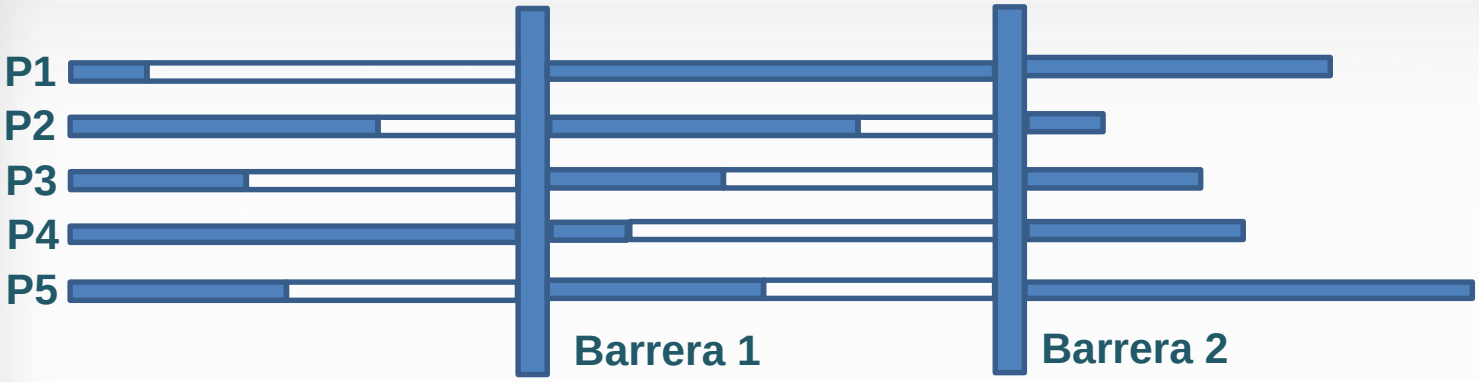
Usado para asegurar que todos los procesos han llegado a un determinado punto.

Es necesario usa un cerrojo, un contador y un flag

Según llegan los procesos incrementan el contador en *exclusión mutua.*

Al llegar todos los procesos se “levanta la barrera” para que puedan continuar procesando.

Sincronización mediante Barreras




```
struct tipo_barrera {
        int S;
        int cont;
        int flag;
    }
```

```
Barrera (B,P)
{
    LOCK (B.S);
        if (B.cont == 0) B.flag = 0;
        B.cont++;
        mi_cont = B.cont;
    UNLOCK (B.S)

    if (mi_cont == P) {
        B.cont = 0;
        B.flag = 1;
    }
    else while (B.flag == 0);
}
```

Consistencia

¿En que orden se ejecutan las instrucciones de un programa?

¿en el que se han programado?

Tanto hardware (segmentación, tomasulo, scoreboard) como software (Optimización por compilación, desenrrolado de bucles, list scheduling…) optimizan la ejecución reordenando código.

Estas reordenaciones funcionan bien en sistemas monoprocesador pero en multiprocesadores pueden dar problemas

Consistencia

- El orden de ejecución en los sistemas paralelos es un importante problema a considerar pues el control de ejecución es distribuido.
- El problema principal es el orden de ejecución de las **instrucciones que acceden a memoria**.
- La **consistencia** trata la correcta imagen que tienen que tener los procesadores teniendo en cuenta el correcto orden de ejecución de las instrucciones de memoria.

Consistencia

¿No es lo mismo consistencia y coherencia?

- La coherencia asegura el correcto cambio de una variable y su correcta propagación.
- También asegura el orden de modificación de esa variable sea igual en todos los procesadores

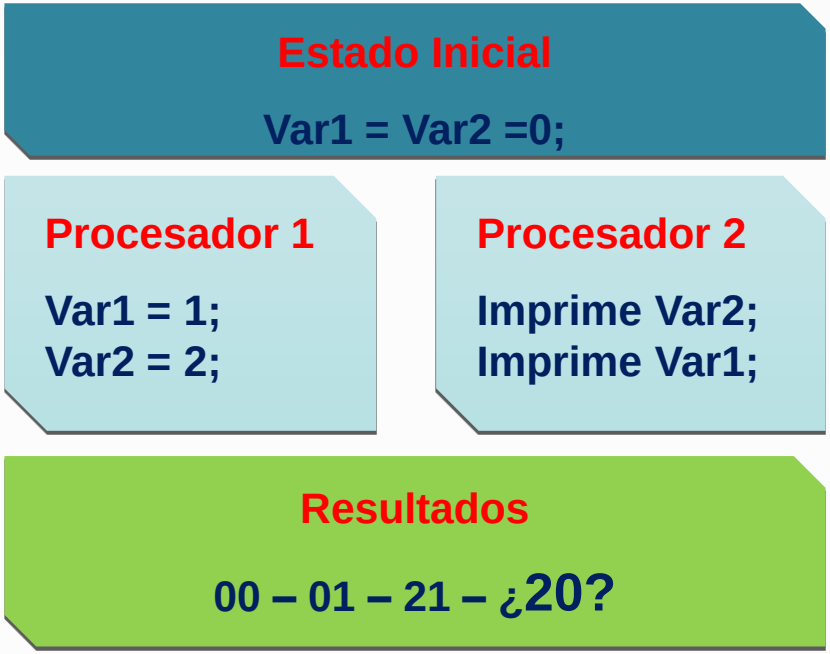
¡¡ La coherencia no asegura el **orden en** que se verán los cambios en **variables diferentes !!**

Este orden en parte es asegurado mediante operaciones de **sincronización**.

La sincronización no es suficiente cuando hay reordenación de código.

Consistencia

Ejemplo:

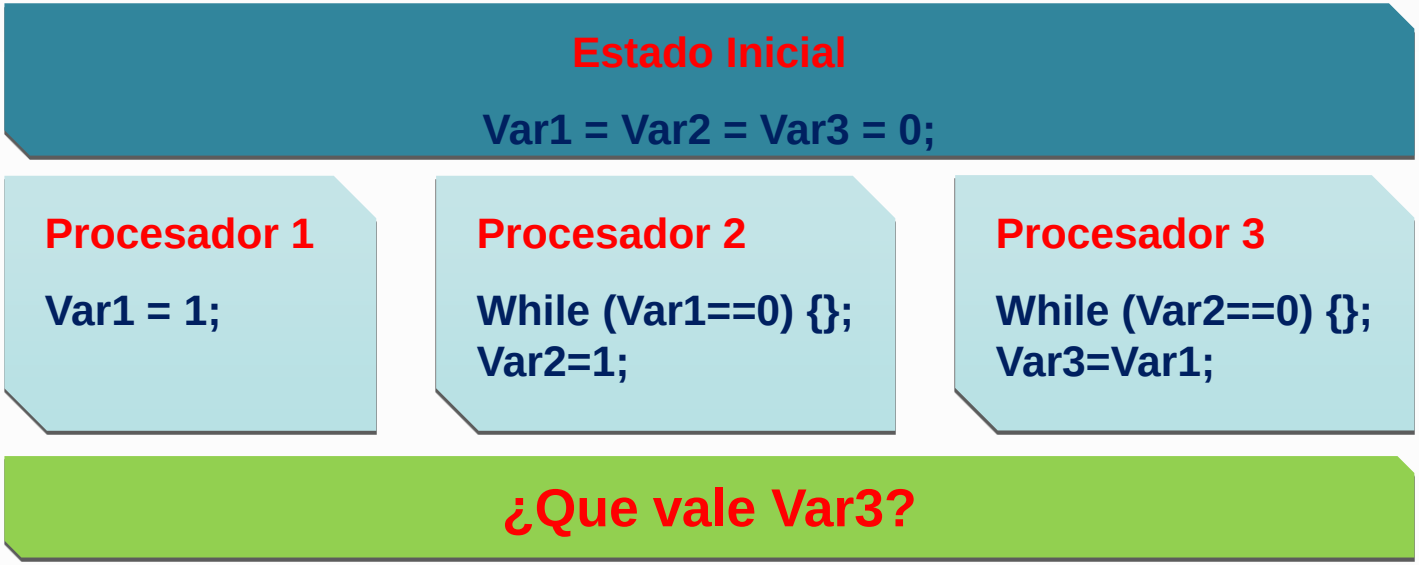


¿Puede estar actualizado var 2 sin estar actualizado var1?

¡El compilador o procesador ha podido hacer una reordenación de código y si podría darse el caso!

Consistencia

Ejemplo 2



La lógica dice que solo puede valer 1, pero la coherencia no asegura que el dato de var1 llegue al procesador3 antes que var2.

Así que puede darse el caso de que var3 acabe con valor 0 en algunas ocasiones

Consistencia

- Un sistema paralelo ha de ser **consistente,** es decir; Los resultados han de ser los mismos que si se ejecuta en un solo procesador.

- Para ello se hace uso de los modelos de consistencia

Consistencia Secuencial (SC)

- Consiste en extender al multiprocesador el modelo de consistencia del monoprocesador. Es decir el orden estricto
- Un multiprocesador es **secuencialmente consistente** si:
  - Mantiene el orden local de las instrucciones
  - El orden global de las instrucciones corresponde a un determinado entrelazado. Necesaria la atomicidad

Este modelo es el más intuitivo, el que normalmente espera un programador.

Consistencia Secuencial (SC)

Mantener el **orden local** implica no desordenar las instrucciones de acceso a memoria.

Accesos P1

VarA;
VarB;

Accesos P2

VarC;
VarD;

Mantienen orden local

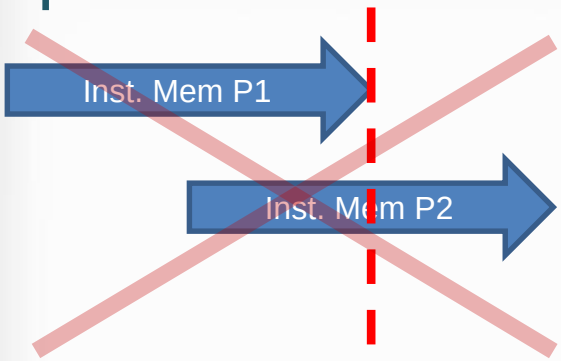
VarA	VarA
VarB	VarC
VarC	VarB
VarD	VarD

NO mantienen orden local

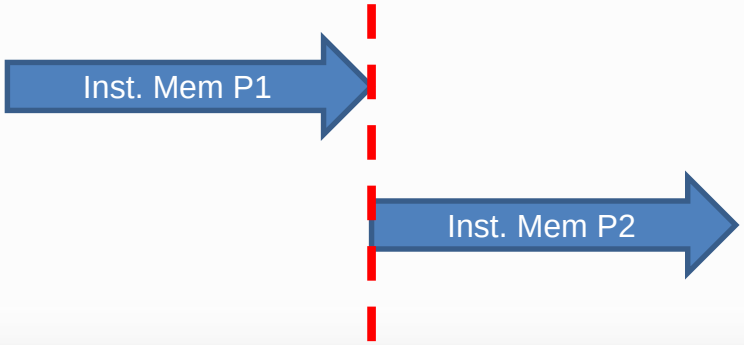
VarA	VarA
VarD	VarD
VarC	VarB
VarB	VarA

Consistencia Secuencial (SC)

Mantener el **orden global** implica que las operaciones en memoria sean atómicas



No se puede comenzar una instrucción a memoria hasta que terminen las anteriores



Consistencia Secuencial (SC)

Orden Global

- La ejecución de una instrucción a memoria ha de esperar hasta las anteriores estén acabadas

¿Cuándo acaba una operación en memoria?

- **¿es de lectura?** al recibir los datos
- **¿es de escritura?** hay que garantizar la escritura y las consecuencias (invalidaciones o actualizaciones)

  ¿Cómo? Completando el protocolo de coherencia con señales de confirmación (líneas de control o mensajes tipo ACK)

Consistencia Secuencial (SC)

- Asegurar el Orden Global es complicado, sobre todo si las copias de cache se han de actualizar y no invalidar.
- La complejidad es tan elevada que apenas se usa la SC en los protocolos de actualización.
- Las restricciones son enormes, **recordemos:**
  - No se pueden hacer dos operaciones en memoria
    (Entre un 25 y un 35% del total)
  - Hemos de asegurar la atomicidad
  - No se pueden usar buffers de escritura
  - El compilador no puede reordenar código
  - No se puede optimar por hardware o segmentación

Modelos de Consistencia Relajados

- Todas las restricciones en SC (*sequencial consistency*) son suficientes para asegurar su funcionamiento pero no todas son necesarias.

- En el modelo secuencial el orden de las instrucciones ha de cumplirse.

Rd→Rd ***Rd→Wr*** ***Wr→Rd*** ***Wr→Wr***

- En los modelos relajados se permite **(¡condicionalmente!)** que no se respete alguno de estos ordenes.

Modelos de Consistencia Relajados

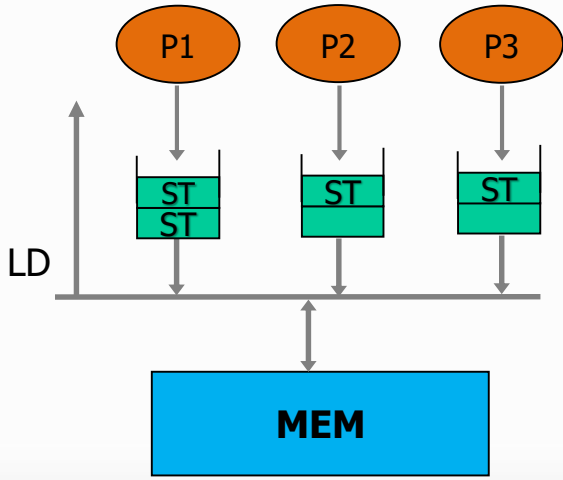
- En un modelo relajado ha de existir siempre la posibilidad de forzar el **orden estricto.**

- Son necesarias nuevas instrucciones de procesador denominadas **barreras de ordenación *(fence)*** utilizadas como puntos de control, estas instrucciones (cada procesador tiene las suyas) son de tipo:
  - ***Write-fence*:** Impone la orden de escritura (Wr→Wr)
  - ***Read-fence*:** Impone la orden de lectura (Rd→Rd)
  - ***Memory-fence:*** Impone todas las ordenes (las 4)

Modelos de Consistencia Relajados
Total Store Ordering (TSO)

- Se permite una lectura aunque no haya finalizado una (o varias) escrituras. Es decir, **no se asegura orden Wr→Rd**.
- **No mantiene la consistencia** por lo ha de tener instrucciones *(fence o RMW)* para poder imponer el orden estricto si fuese necesario

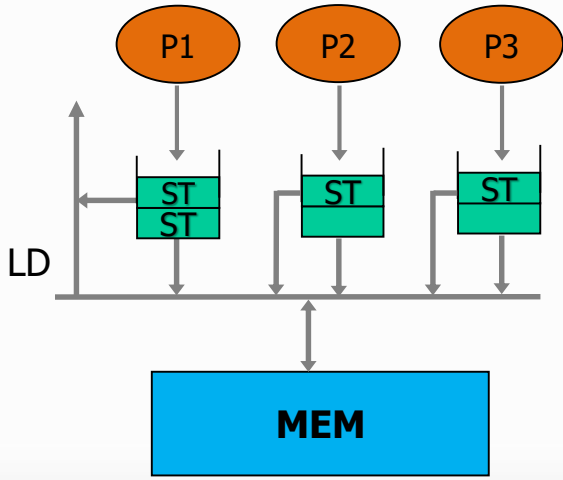
- Los ST se van encolando en buffers por procesador
- Colas FIFO
- Los ST han de mantener el orden (entre STs)
- Los LD pueden adelantar los ST
- Los LD Han de mantener el orden (entre LDs)



Modelos de Consistencia Relajados
Partial Store Ordering (PSO)

- Menos restrictivo que el anterior. **no se asegura orden Wr→Rd ni el Wr→Wr.**
- **No mantiene la consistencia** por lo ha de tener instrucciones *(fence o RMW)* para poder imponer el orden estricto si fuese necesario

- Implementación similar al anterior pero las colas no son FIFO por lo que una ST puede adelantar a otra ST



Modelos de Consistencia HiperRelajados
Weak Ordering (WO)

- No impone ningún orden salvo las operaciones sincronizadas que activa un ***fence*** total.
  - Al encontrar una sincronización hay que esperar la finalización global de las operaciones en memoria
  - Las operaciones en memoria posteriores al área sincronizada también tendrán que esperar a que todas las operaciones sincronizadas acaben

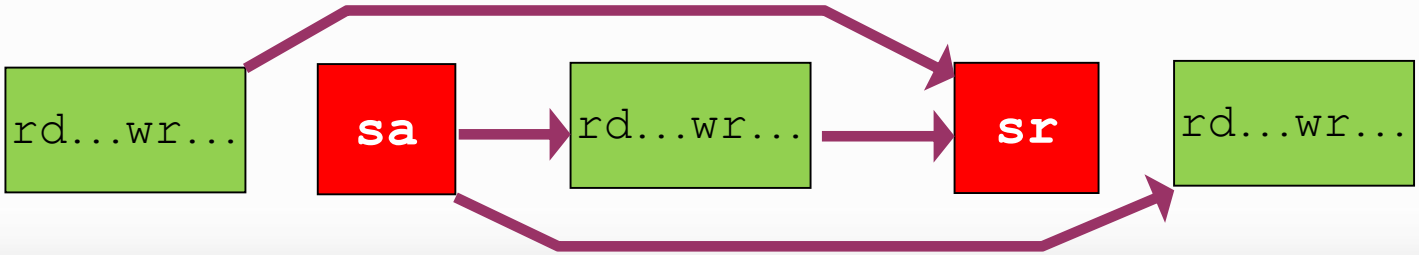
rd / wr → s s →rd / wr s → s.

Modelos de Consistencia HiperRelajados
Release Consistency (RC)

- Más flexible aún que el anterior.
- No se exige el orden dentro de las partes sincronizadas.

- El sincronismo se divide en dos
  - Adquisicion (sa-acquire)
  - Liberacion (sr-release)

sa → rd / wr **rd / wr → sr**



Resumen Modelos de Consistencia