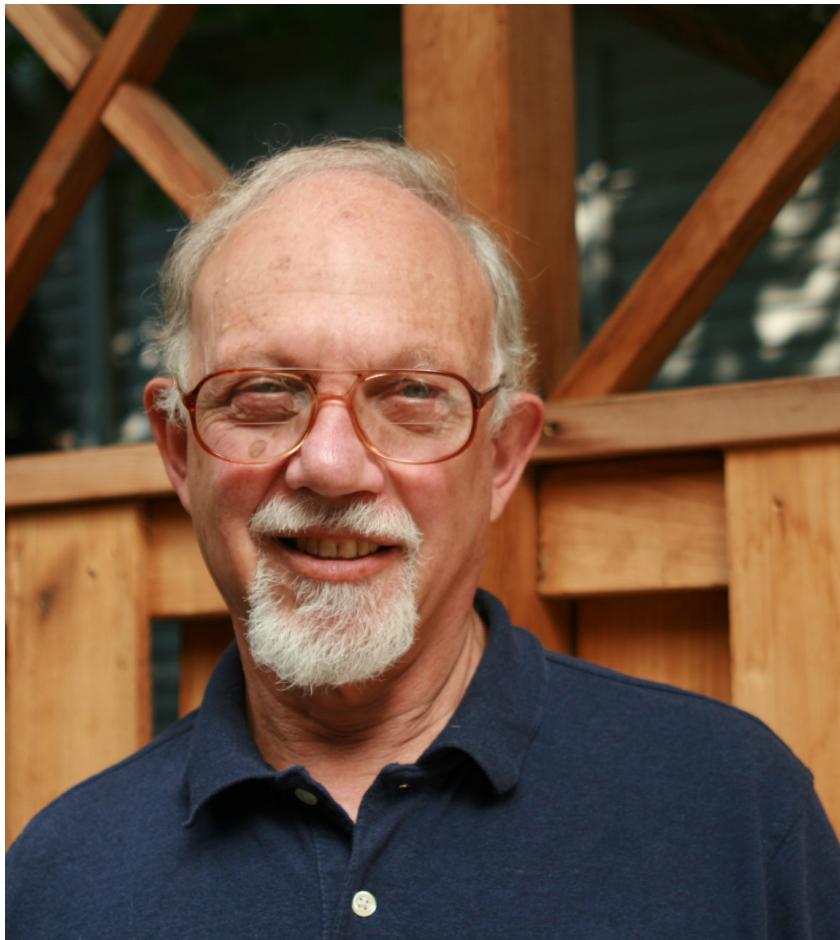


# Sincronização

José Nelson Amaral, U of Alberta  
(adaptado por Guido Araujo)

# Livro Texto



Jean-Loup Baer – University of Washington

# A Necessidade de Sincronização

// Initially A == 0

Process P1:

$A \leftarrow A + 1$

Process P2:

$A \leftarrow A + 2$

Qual o valor de A depois que os processos P1 e P2 executam?

Manter as caches coerentes resolve este problema?

A Não // O

Não, a invalidação de A vinda de P2 pode chegar em P1 depois do valor ter sido carregado em R1

Process P1:

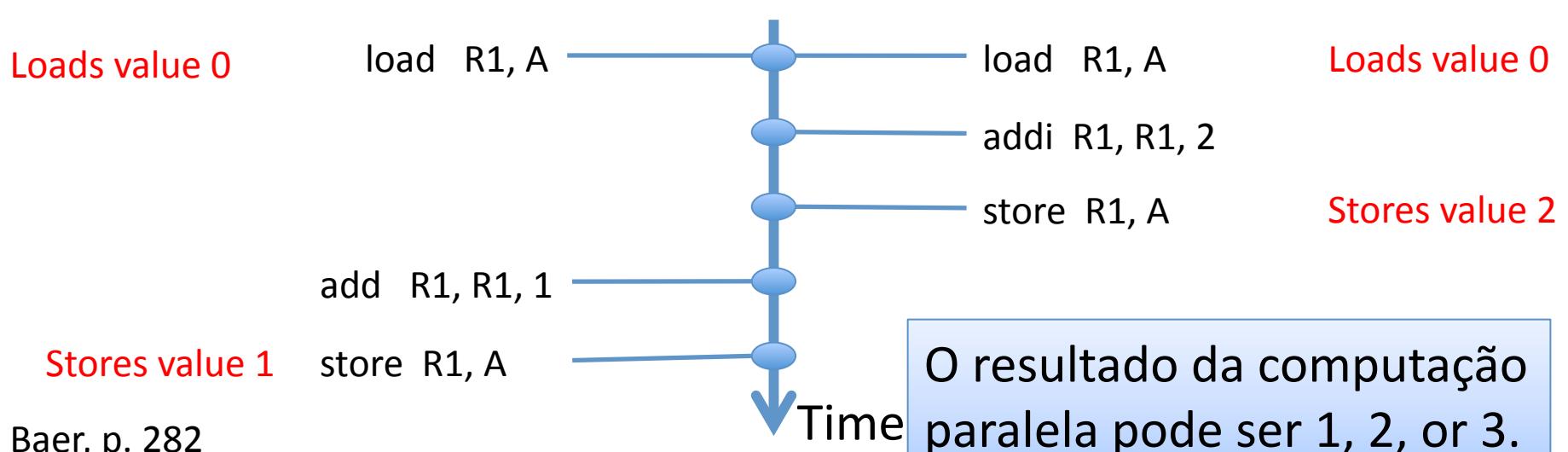
$A \leftarrow A + 1$

load R1, A  
addi R1, R1, 1  
store R1, A

Process P2:

$A \leftarrow A + 2$

load R1, A  
addi R1, R1, 2  
store R1, A



Nós precisamos de uma lock!

# Código Sincronizado

// Initially A == 0

Process P1:

```
while (! acquire(lock)) wait();
A ← A + 1
release(lock);
```

Process P2:

```
while (! acquire(lock)) wait();
A ← A + 2
release(lock);
```

Qual é o valor de A depois que os processos P1 e P2 executam agora?

Agora A certamente tem o valor 3.

# Código Sincronizado

```
// Initially A == 0
```

Process P1:

```
while (! acquire(lock)) wait();  
A ← A + 1  
release(lock);
```

Duas opções para *wait()*:

Process P2:

```
while (! acquire(lock)) wait();  
A ← A + 1  
release(lock);
```



Como  
acquire(lock)  
funciona?

***busy wait*** or ***spin lock***: o processo continua executando enquanto espera

***blocking***: o processo suspende a execução enquanto espera, libera a CPU e é acordado quando a lock é liberada.

# Problema de Aquisição de Lock

- “lock” é simplesmente uma variável armazenada em memória compartilhada 
- A lock pode ter dois valores:
  - 0 (lock está livre)
  - 1 (lock está adquirida)

Complicação: tem que prevenir outros processadores de modificar a lock entre test e set
- Passos para adquirir uma lock:
  - testa se o valor é 0 ou 1
  - se o valor é 0, muda para 1 para adquirir a lock



Usa uma operação indivisível: *test-and-set*.

# Instrução *test-and-set*

- ISA garante que não existe nenhum outro acesso à variável entre *test* e *set*



1964: a IBM já tinha uma instrução test-and-set no seu mais novo computador: o IBM 360.

Hoje todo processador tem uma operação atômica.

# Variações do *test-and-set*

- *exchange-and-swap*: Permite outros valores além de 0 e 1
  - troca os valores entre um registrador e uma posição de memória
- *compare-and-swap*: a memória só é modificada se ela contém o valor do teste especificado.



Intel Architecture 32



SPARC

# *fetch-and-Θ*

- O carregamento de um valor de uma posição da memória e a realização de uma operação na mesma posição são indivisíveis
- Exemplos:
  - fetch-and-increment
  - fetch-and-add
  - fetch-and-store

# Usando *fetch-and-Θ* no exemplo

// Initially A == 0

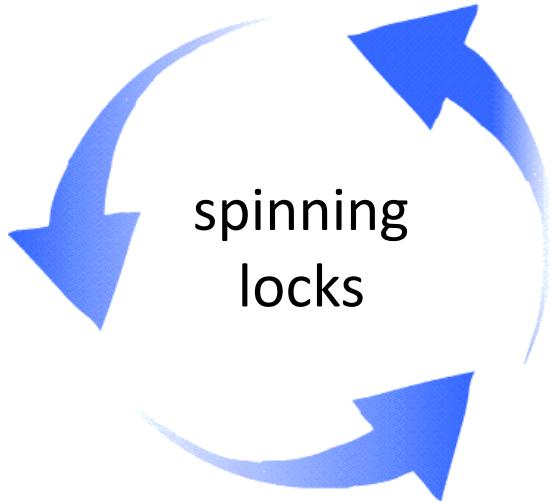
Process P1:

fetch-and-increment A

Process P2:

loadi, R1, 2  
fetch-and-add A, R1

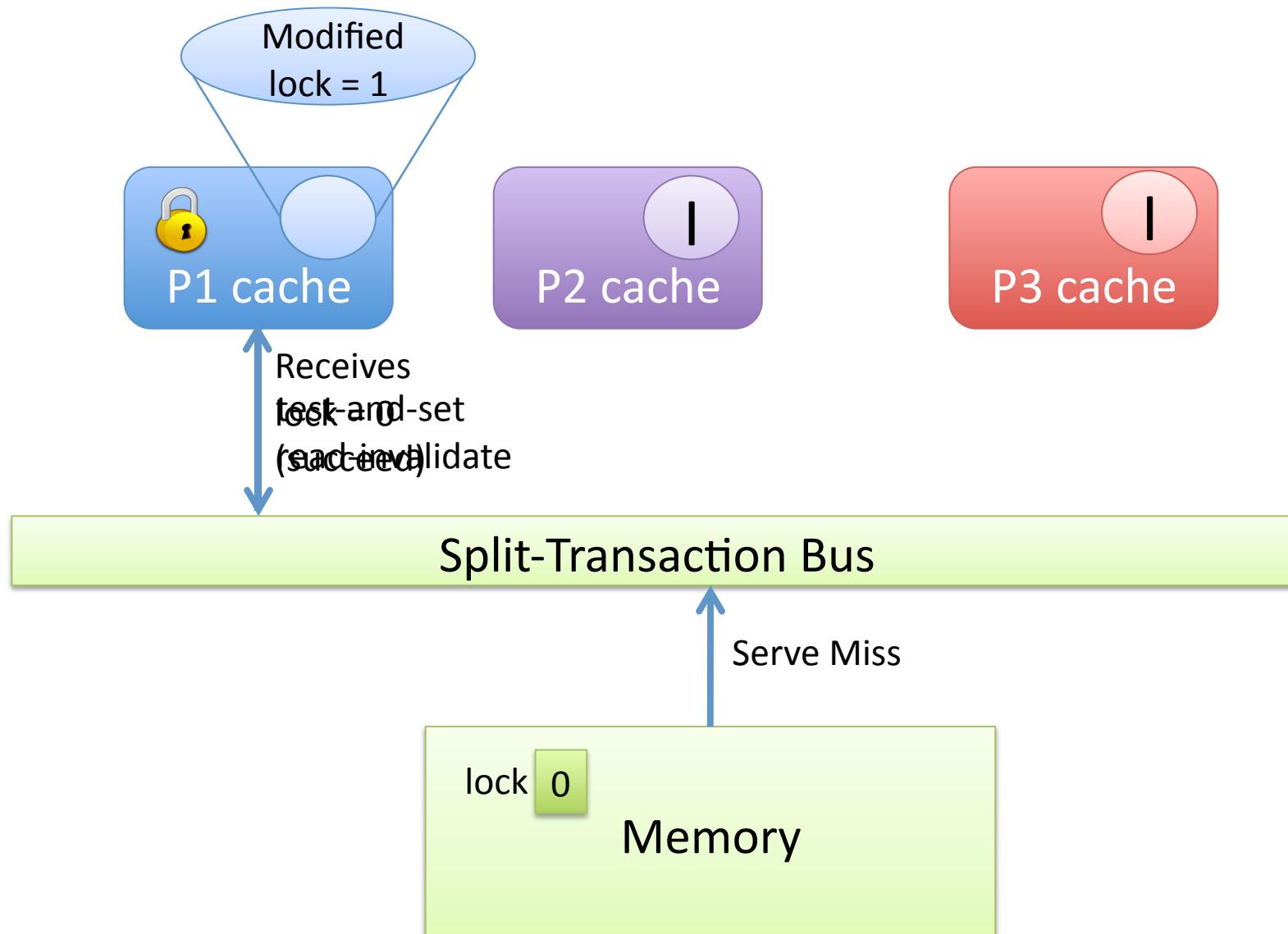
# Contenção por Locks em test-and-set



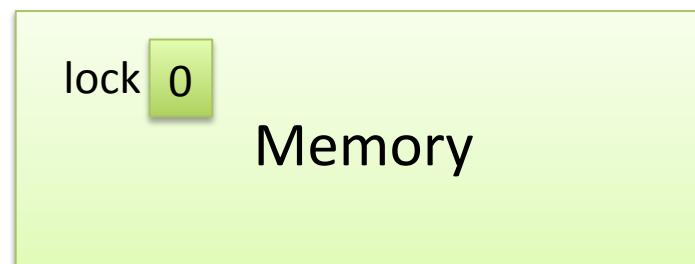
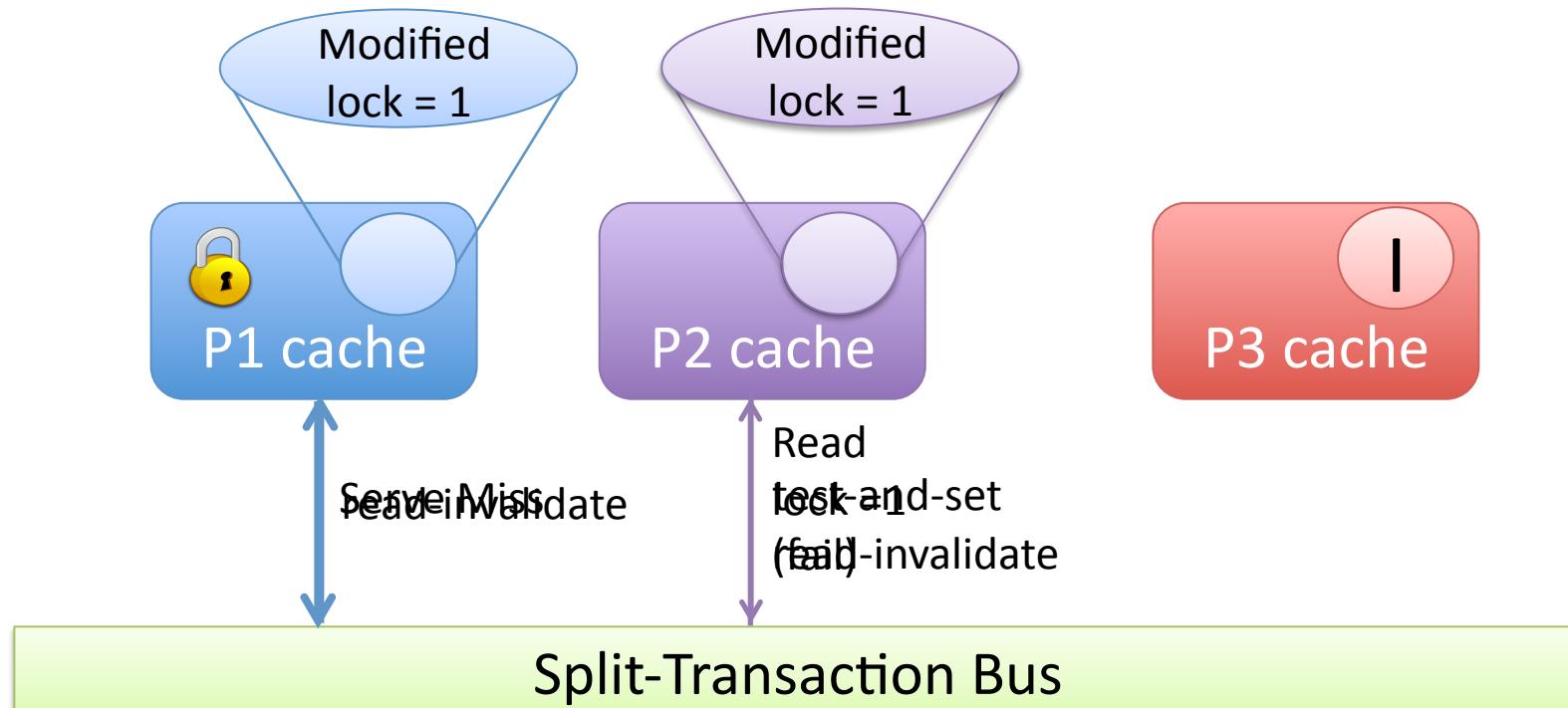
Repetidamente tenta adquirir uma lock que um outro processador possui.

**Problema:** test-and-set sempre invalida a posição de memória nas outras caches.

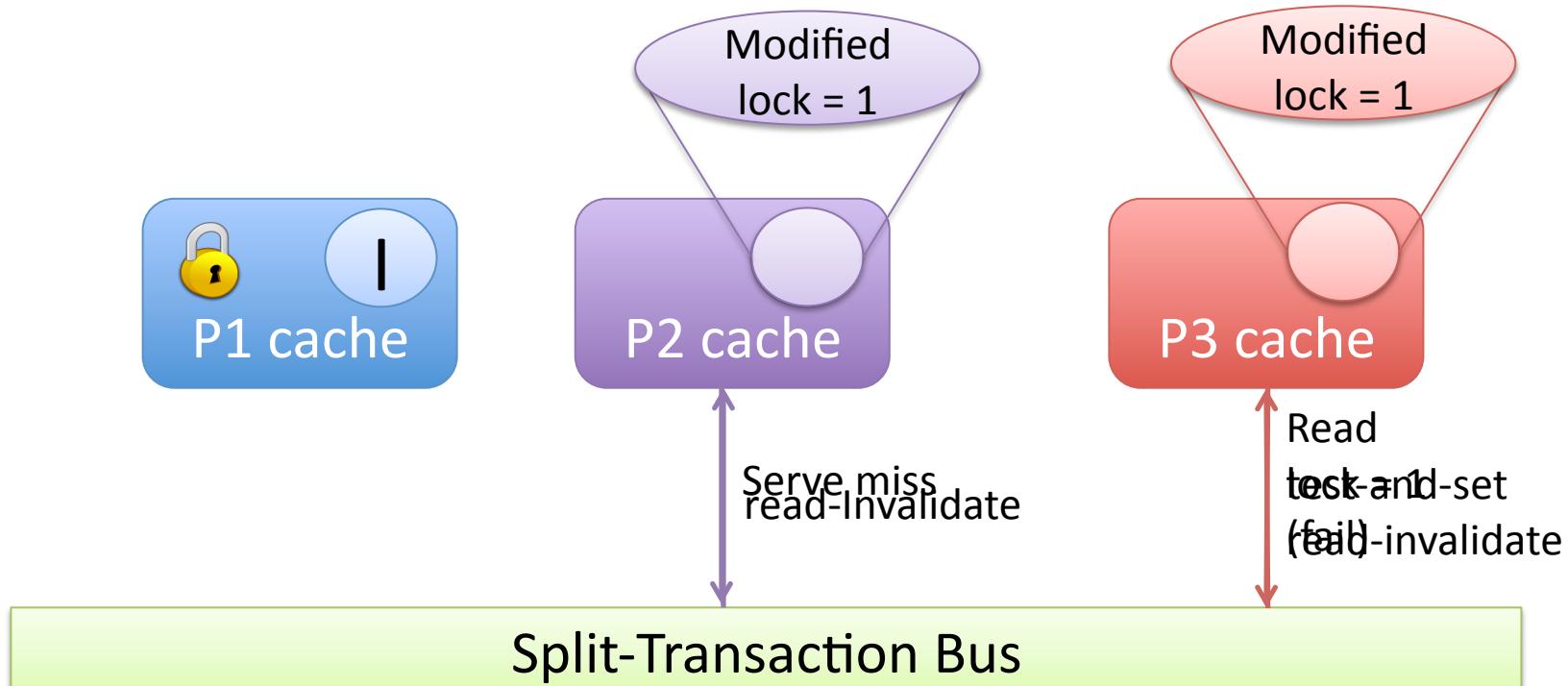
# Example



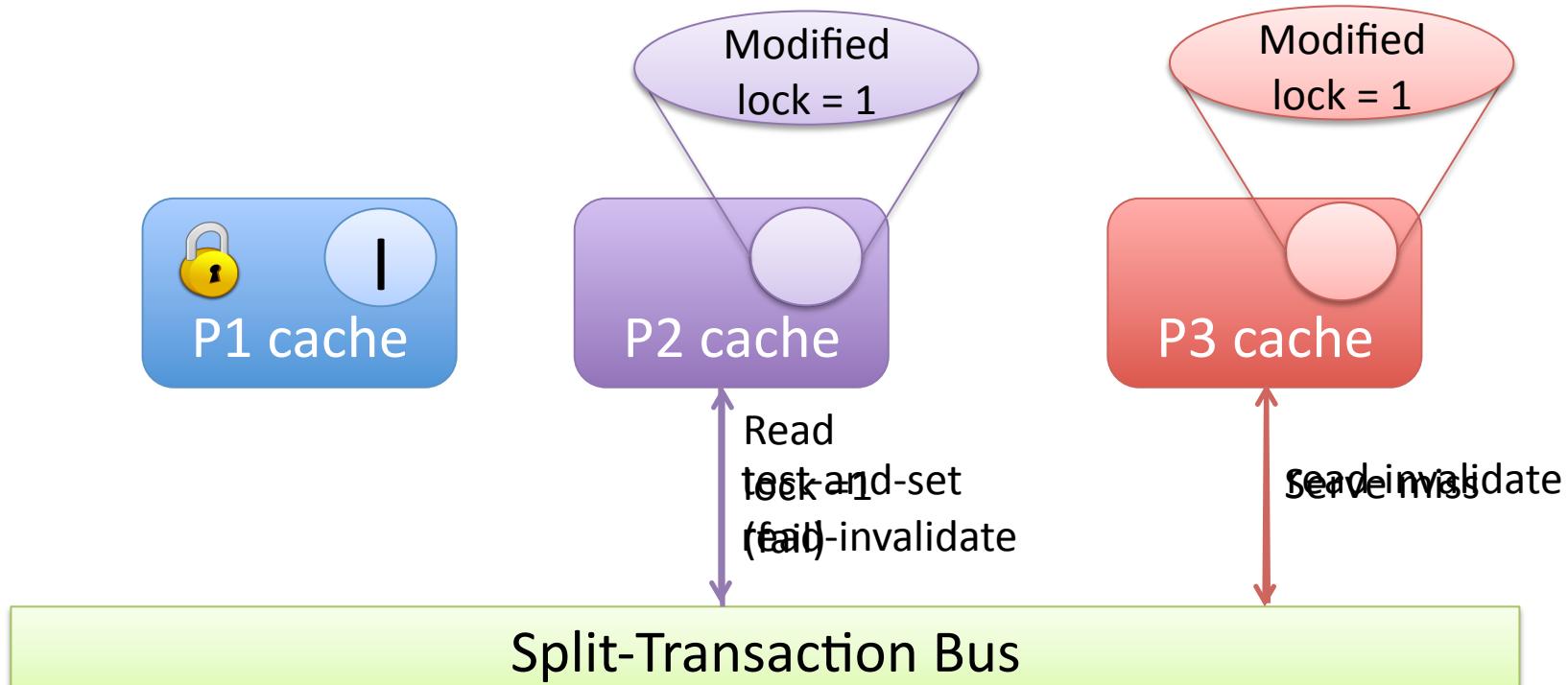
# Example



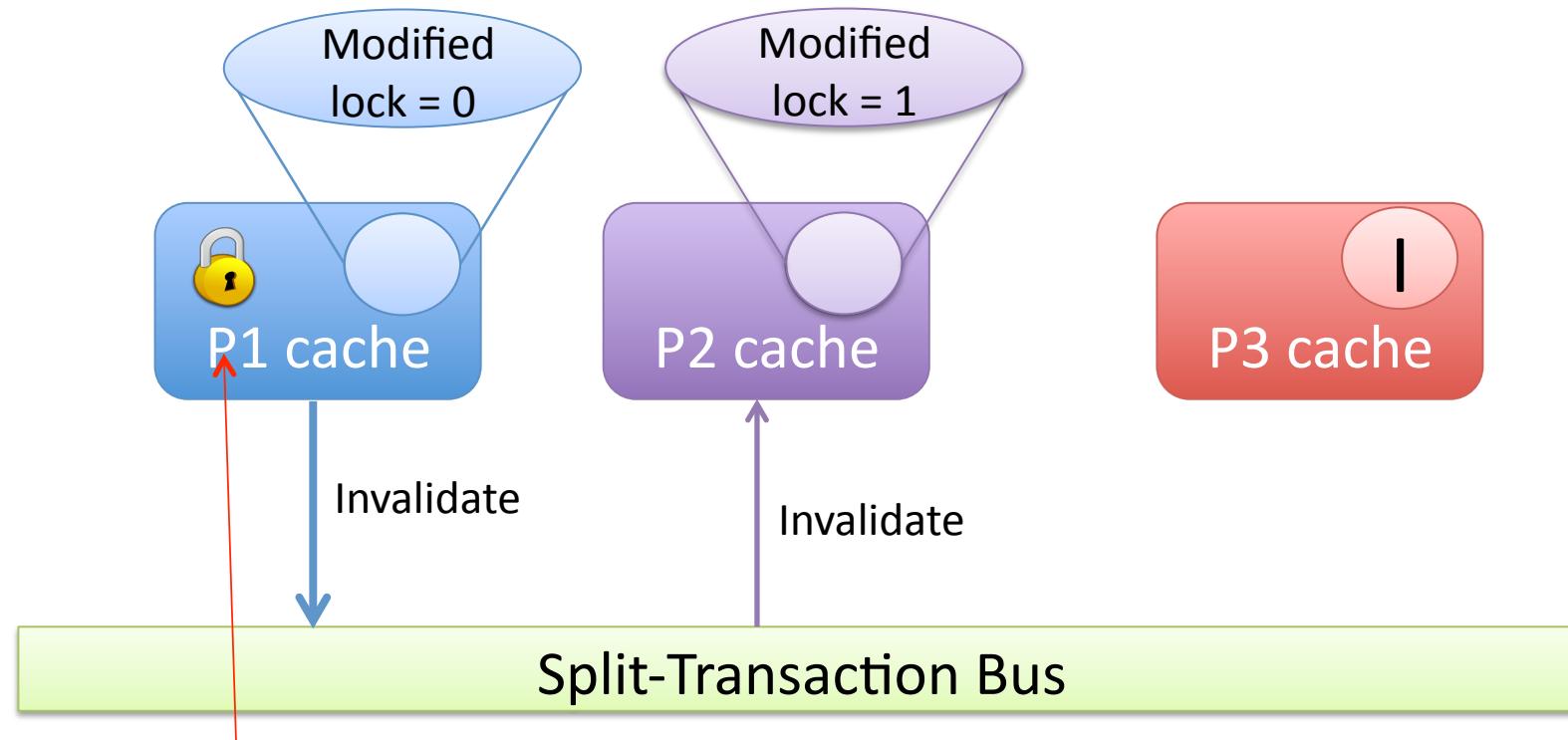
# Example



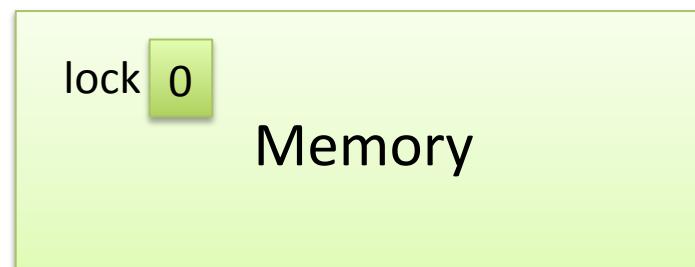
# Example



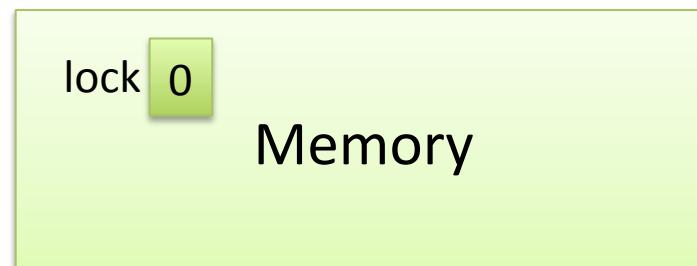
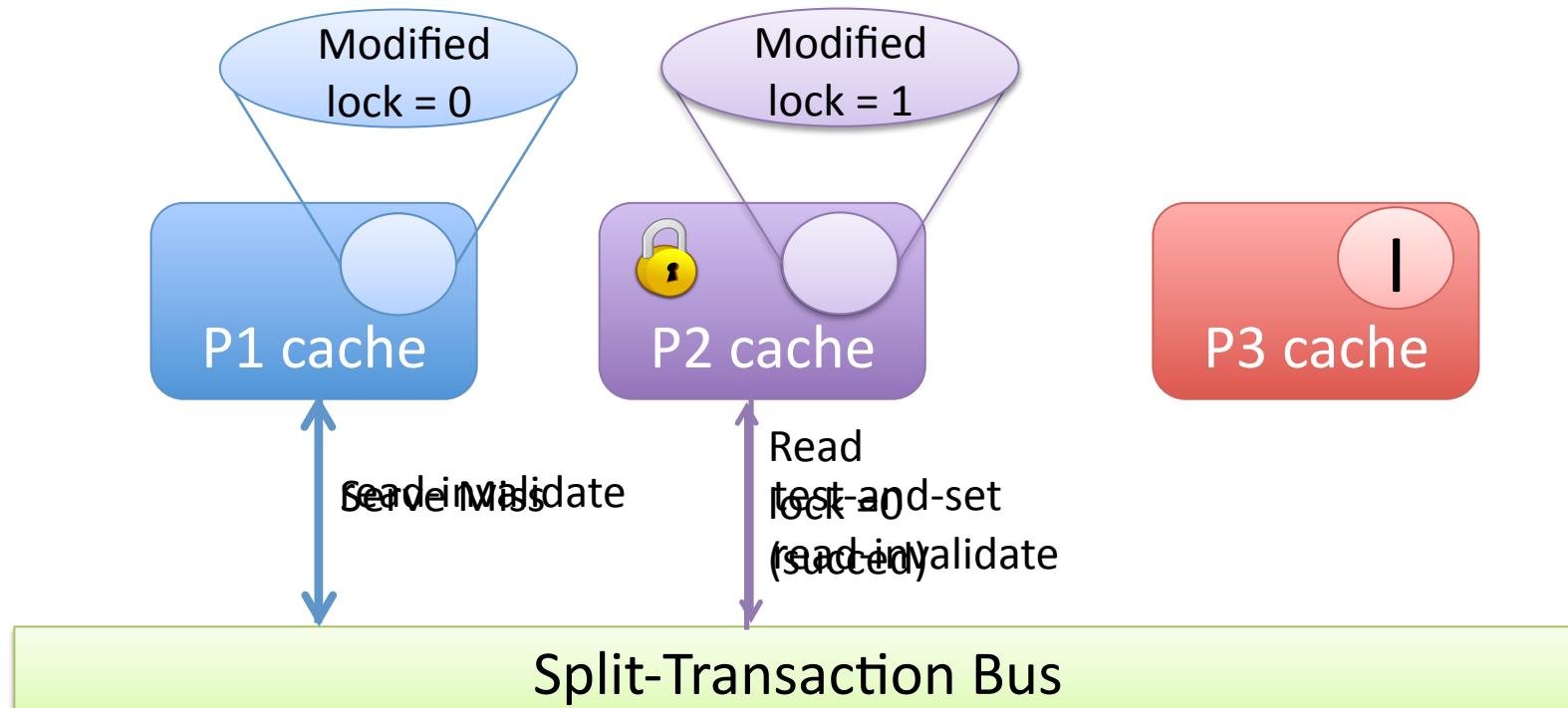
# Example



Para liberar a lock, este processador terá que competir pelo barramento com os processadores que estão spinning.



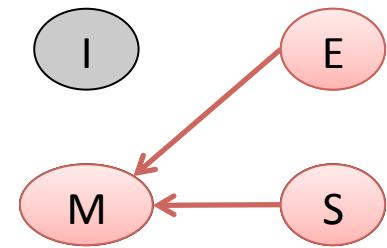
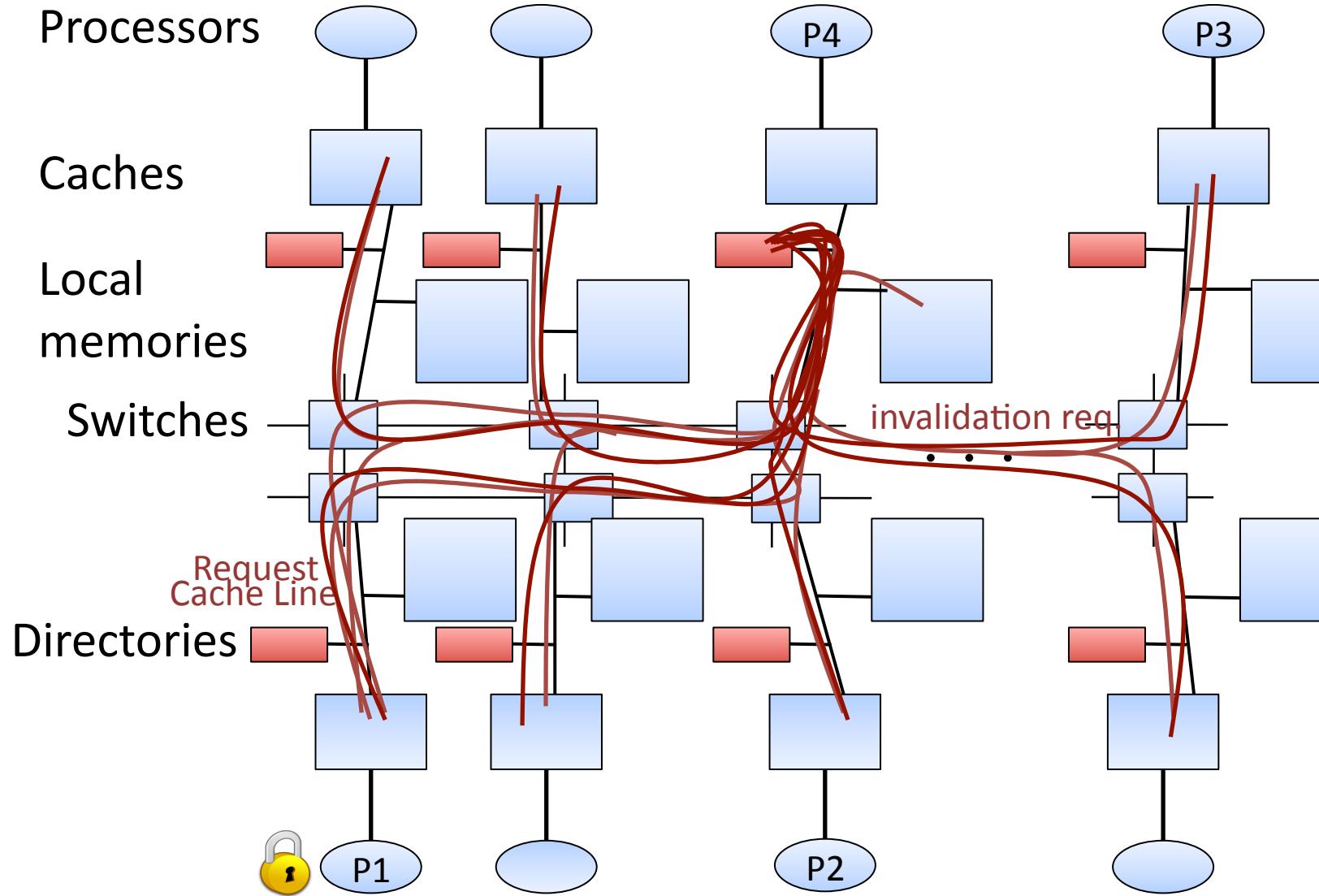
# Example



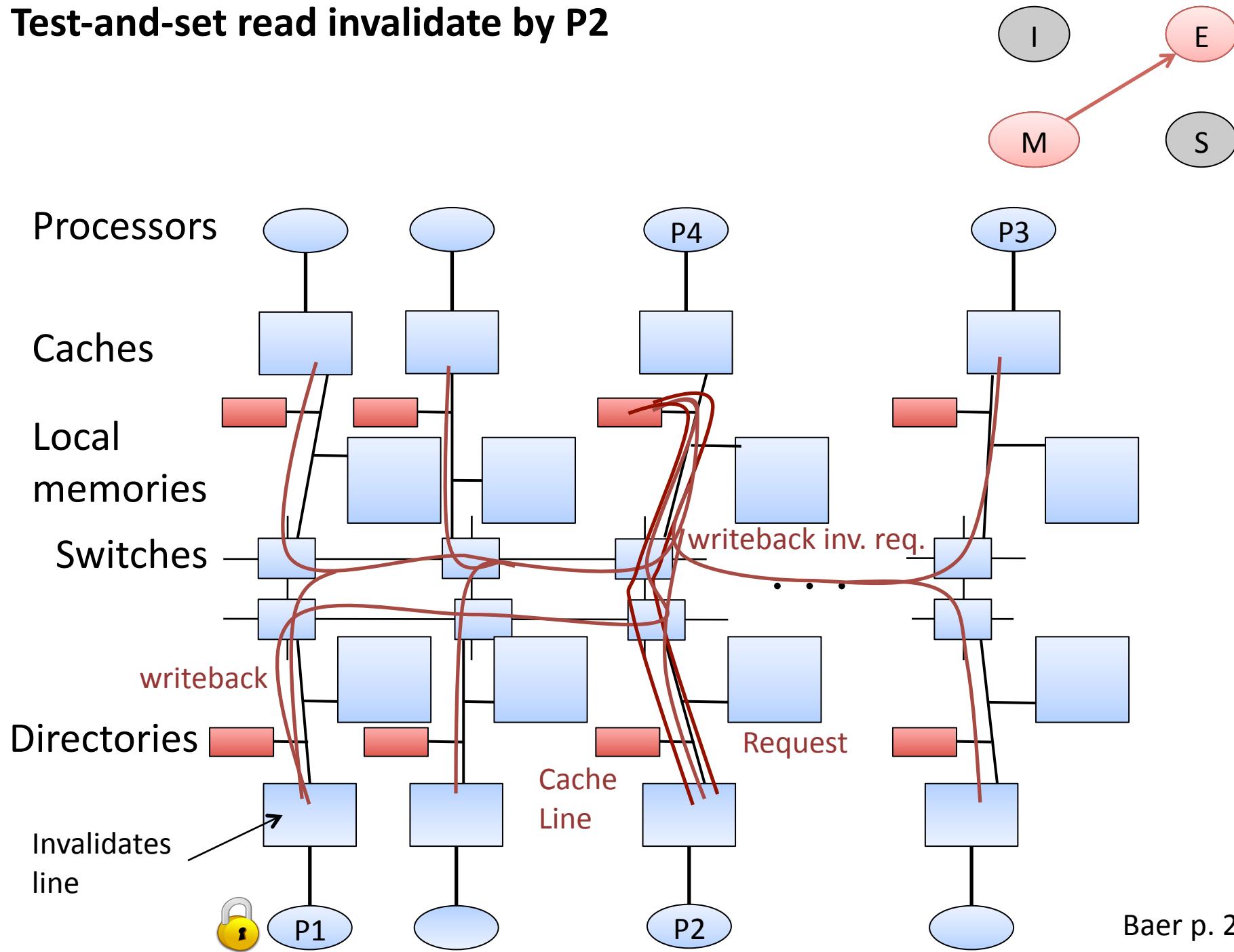
Muito tráfego no barramento ou um ponto quente na rede de interconexão.

# Test-and-set read invalidate por P1 para uma linha no estado Exclusive ou Shared:

all caches must acknowledge



## Test-and-set read invalidate by P2



# Starvation

P1 cache

P2 cache

P3 cache

Split-Transaction Bus

Se P2 adquire a lock depois de P1 e P1 adquire a lock depois de P2 e assim por diante, então P3 ficará faminto (*starve*).

# Back-off exponential

Alivia contenção e o risco de starvation.

P1 cache

P2 cache

P3 cache

Split-Transaction Bus

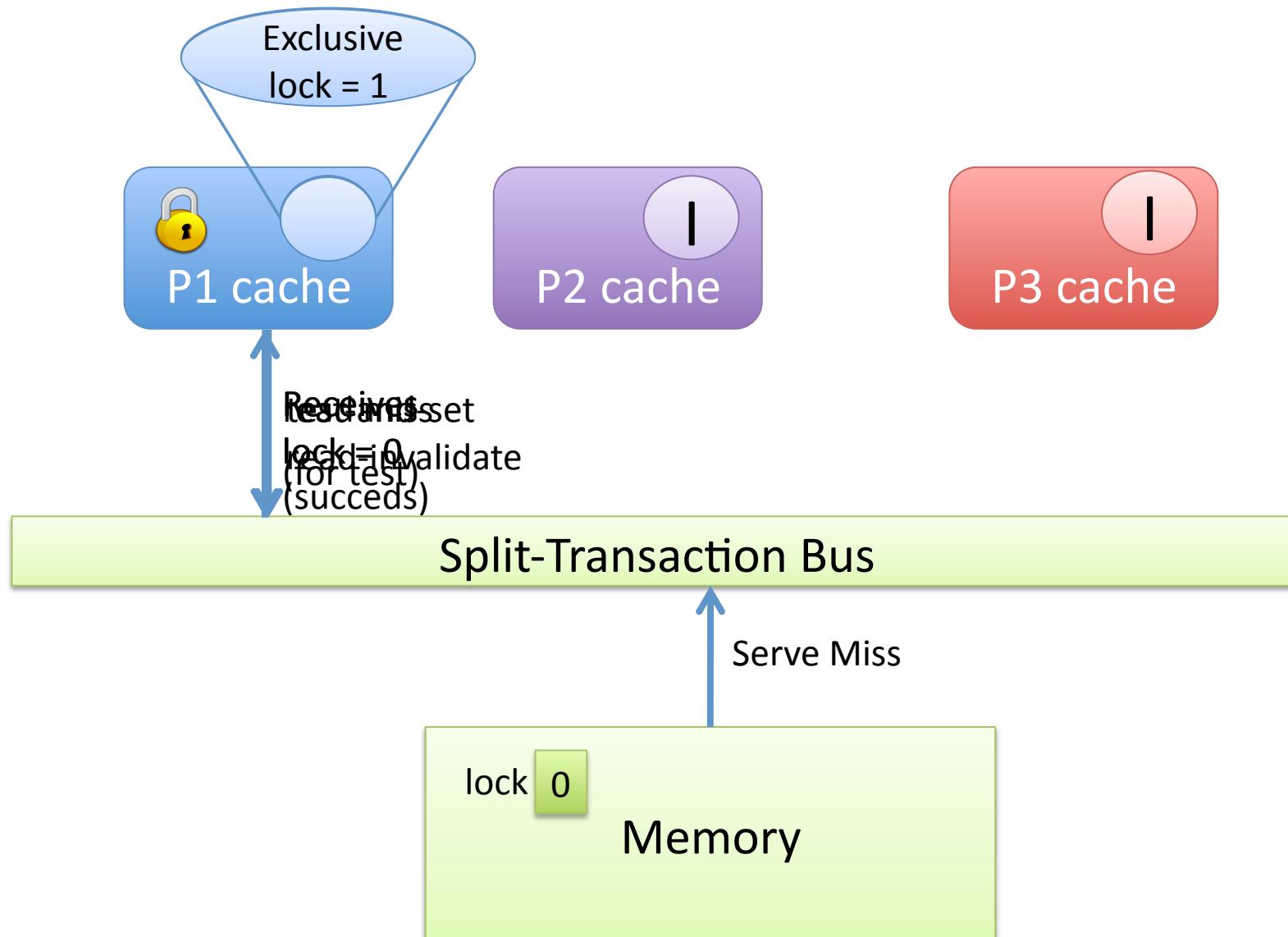
Quando P2 tenta adquirir a lock e falha, P2 espera por um tempo  $t$  antes de tentar de novo.

Se P2 falha de novo, P2 espera por um tempo  $2t$  antes de tentar de novo.

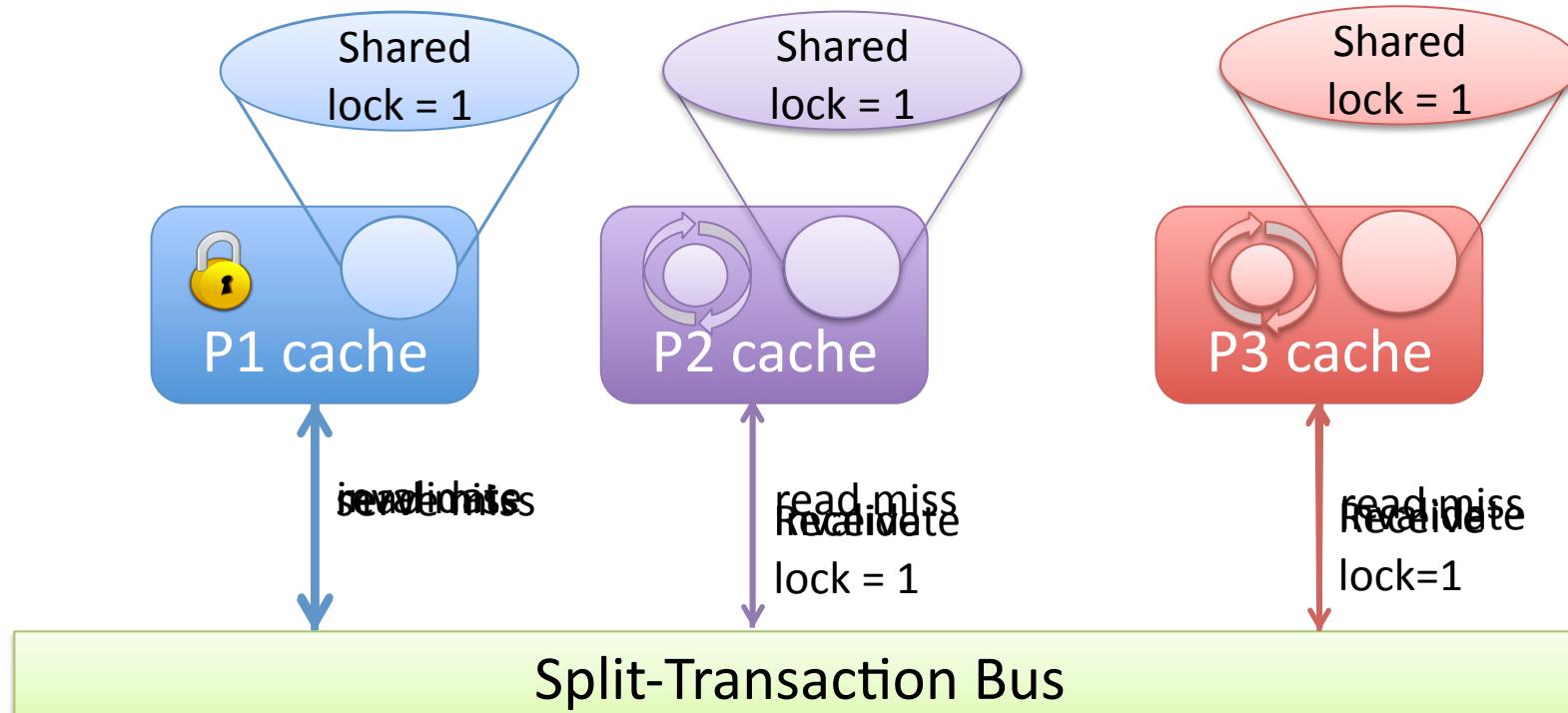
Depois espera por  $4t$ ,  $8t$ , e assim por diante.

Reducindo contenção por Locks  
test-and-test-and-set

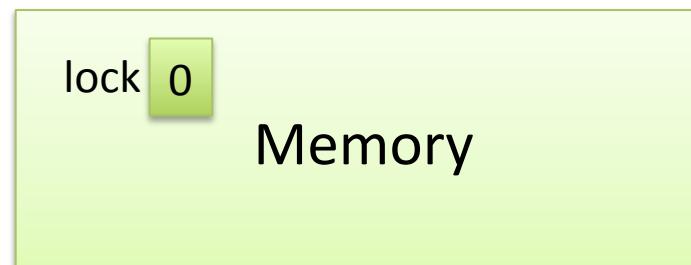
# *test-and-test-and-set*



# *test-and-test-and-set*

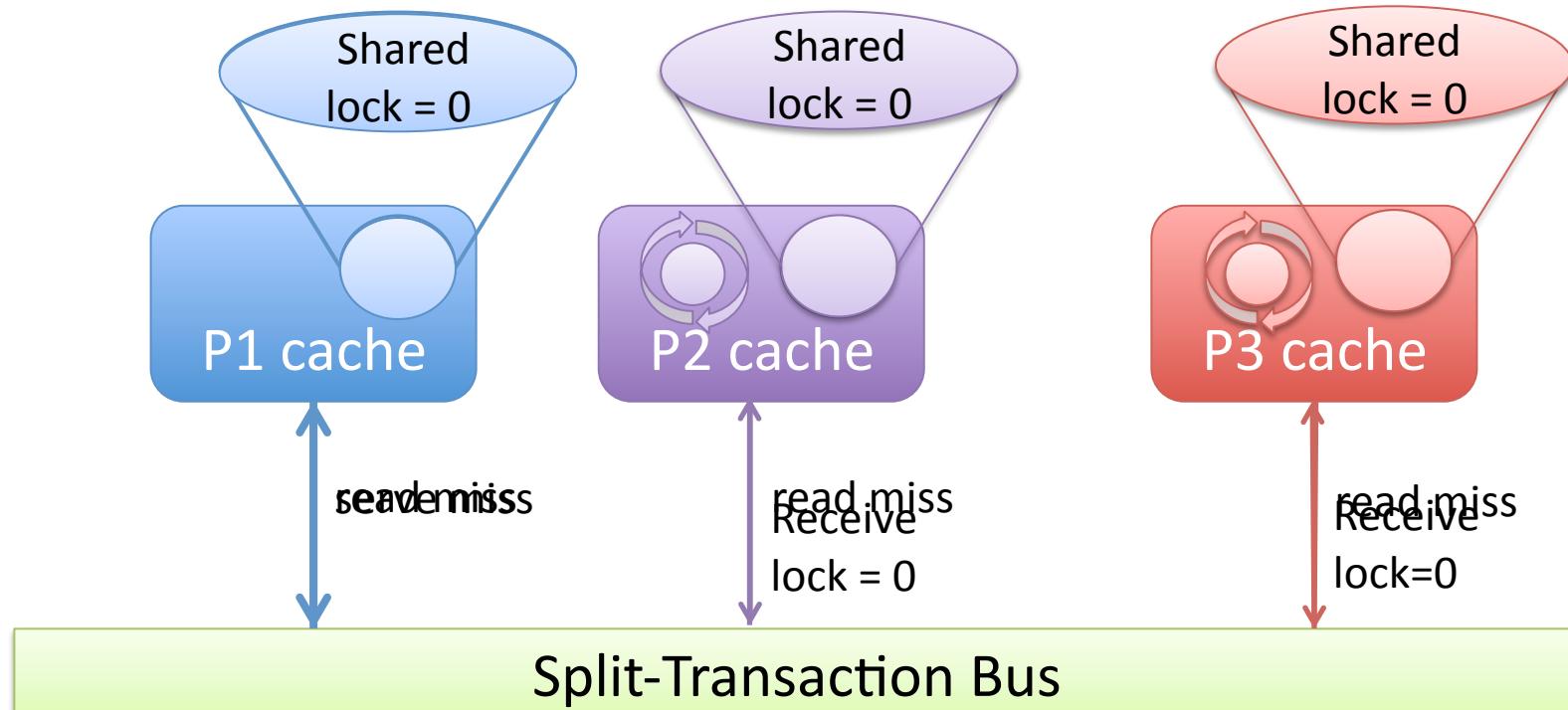


que acontece quando o processador que possui a lock (P1 neste exemplo) libera a lock?

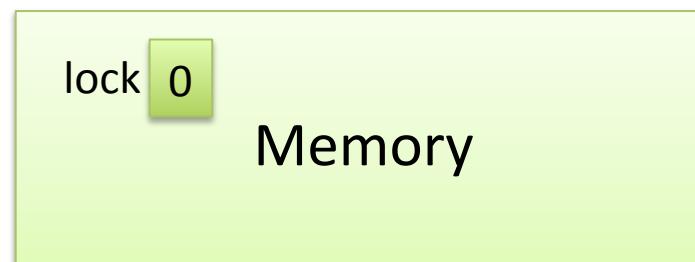
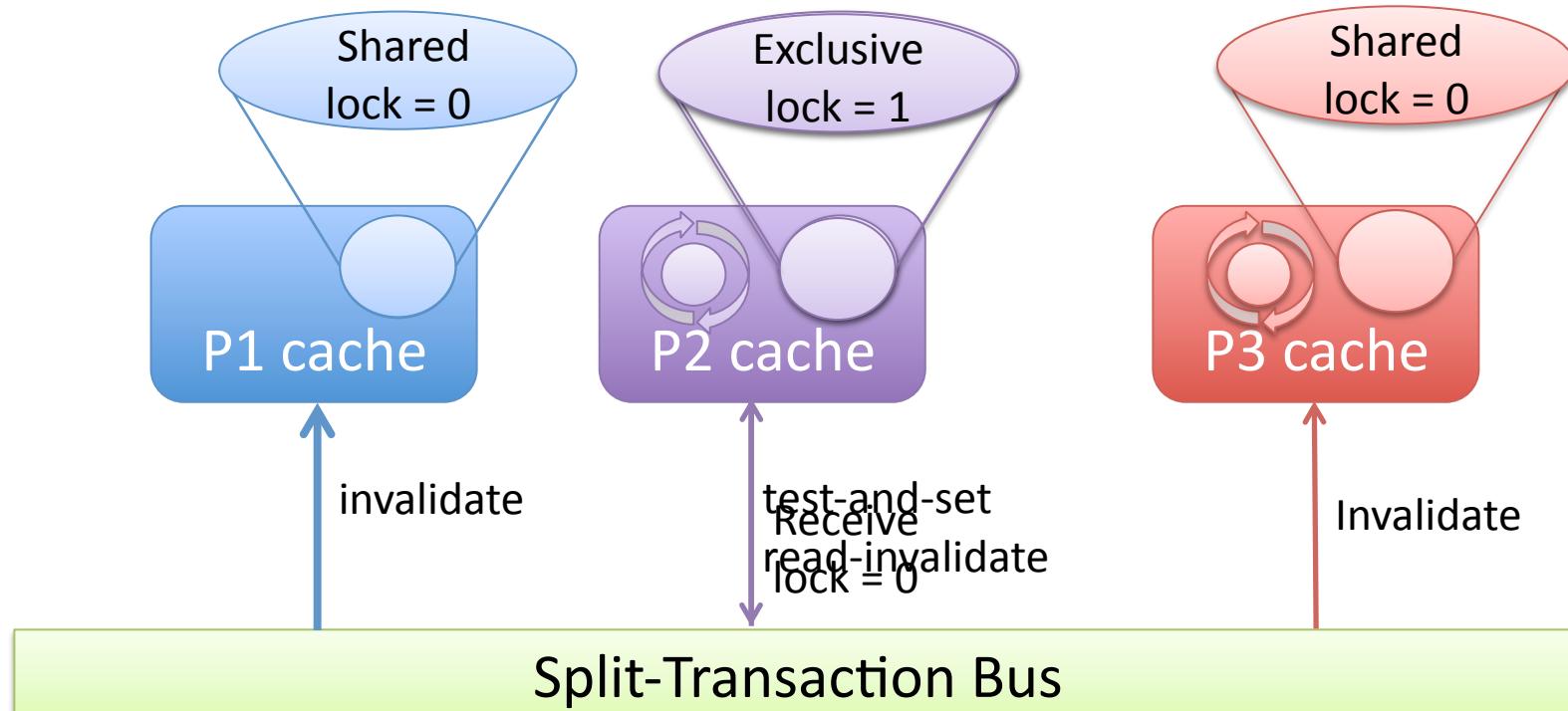


A competição pela lock será pior do que antes.

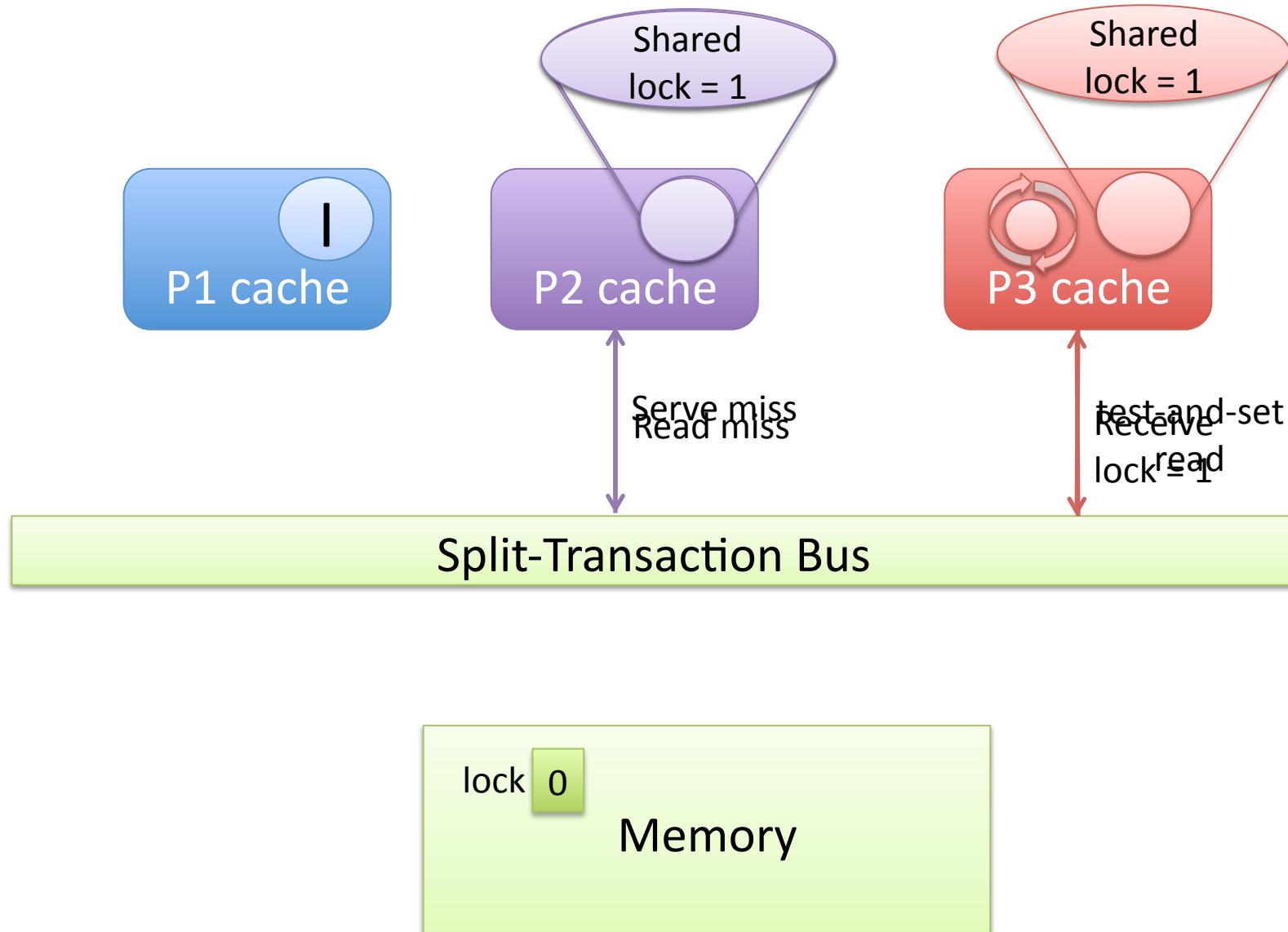
# *test-and-test-and-set*



# *test-and-test-and-set*



# *test-and-test-and-set*



# Contenção por Lock

## Ciclo de Realimentação Positiva

Um processador P1 adquire uma lock para executar uma curta sessão de código que acessa variáveis compartilhadas.

Depois de adquirir a lock, P1 compete pelo barramento/rede com os outros processadores que ainda estão rodando o protocolo de lock.

Quanto mais contenção, mais longa é a sessão crítica.

Uma sessão critica longa aumenta o número de processadores que estão competindo pela lock.

# Eliminando starvation

# Filas de Espera Para Locks

Uma fila de espera (distríbuida) em hardware para processadores

- enqueue:** 1. aloca uma cópia da linha que contém a lock na cache do processador.  
2. entra o processador na fila  
3. Se a lock está ocupada, spin na cache local

**dequeue:** ocorre quando o processador que segura a lock

1. libera ela
2. manda a lock liberada e os dados da mesma linha para o próximo processador que está esperando

Complicações:

- requer transferências diretas de uma cache para outra.
- mais estados para coerência na cache.
- tem que executar algumas operações fora do protocolo de coerência.

# Filas de Espera Para Locks

```
init: flag[0] := 0;                                /*initially; 1st processor can have the lock*/
      for (i :=0; i < n; i++)
          flag[i] := 1;
      tail := 0;
acq: myindex := fetch-and-increment (tail);    /* the increment is modulo n*/
      while (myindex = 1);                          /*spin while the lock is held elsewhere*/
      flag[myindex] := 1;                           /*I got the lock. Make it busy for next round*/
rel: flag[(myindex + 1) mod n] :=0                 /* release the lock and pass it on to the next processor*/
```

Exemplo de *test-and-test-and-set*  
*load-locked* e *store-conditional*

# Sincronização na arquitetura MIPS

- Load linked:
- Store conditional:

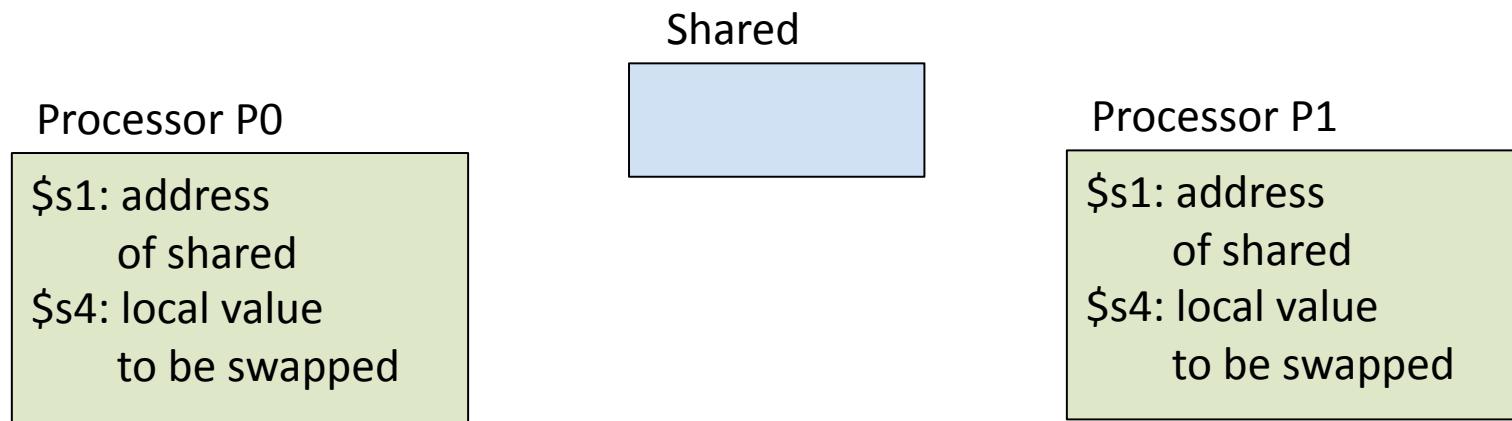
ll rt, offset(rs)

sc rt, offset(rs)

- Sucesso se a posição não mudou desde o último ll
  - Retorna 1 em rt
- Falha se a posição mudou
  - Returns 0 em rt
- Exemplo: atomic swap (para testar/setar uma lock)

# Exemplo: (atomic swap)

Usada para testar/setar uma variável



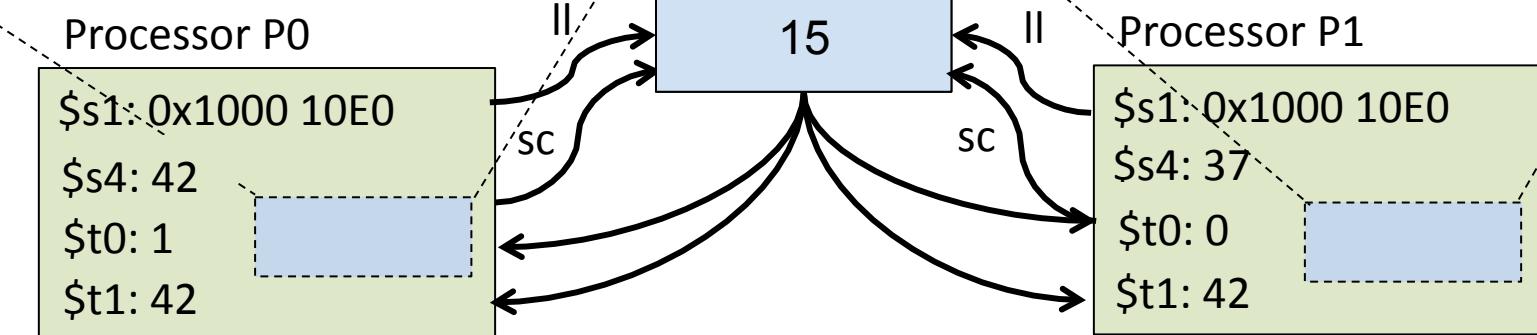
Linked  
Register

# Exemplo: (atomic swap)

Linked  
Register

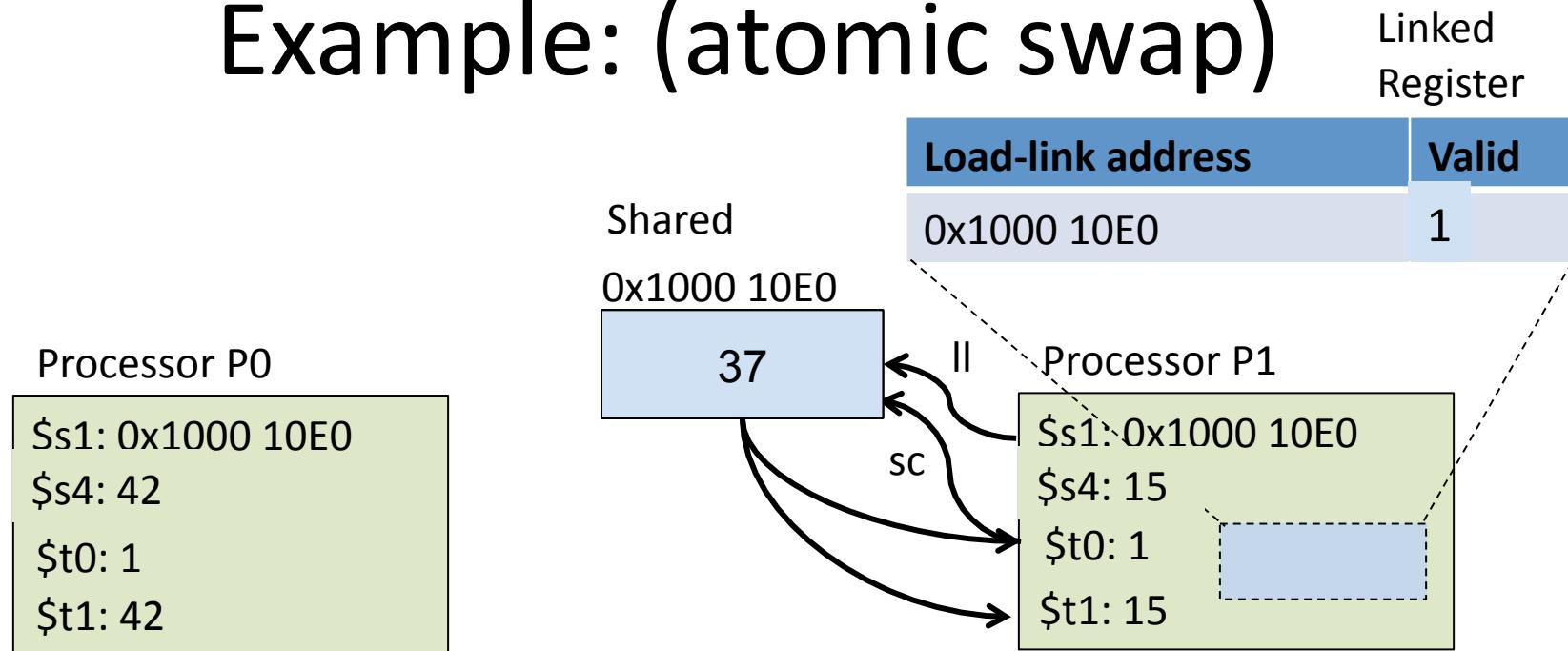
Load-link address	Valid
0x1000 10E0	1

Load-link address	Valid
0x1000 10E0	0



```
try: move $t0, $s4      # copy exchange value
```

# Example: (atomic swap)



```

→ try: move $t0, $s4      # copy exchange value
        ll      $t1 , 0($s1)  # load linked
        sc      $t0 , 0($s1)  # store conditional
        beq    $t0 , $zero,try # branch store fails
        move   $s4 , $t1      # put load value in $s4
    
```

# Full-Empty Bit

Um bit para sincronização em cada posição de memória.

bit = 0: nenhum valor foi escrito na posição.

Uma leitura (load) entra em espera.

Uma escrita (store) continua e seta o bit para 1.

bit = 1: um valor foi escrito

Uma escrita (store) entra em espera.

Uma leitura (read) continua e reseta o bit para 0.

Muito útil para sincronizacao produtor-consumidor.

# Modelos de Memória

O que o programador espera?

# Modelos de Memória

```
// initially flag = 0
```

Process P1:

```
write (A);  
flag ← 1
```

Process P2:

```
while (flag != 1); /* spin on flag */  
read (A);
```

Qual é a intenção do programador?

P1 produz o valor de A para ser consumido por P2

É possível que P2 leia o valor antigo de A?

Sim, pois não existe dependência entre A e flag.

O compilador pode reordenar as duas escritas.

O hardware pode reordenar as duas escritas.

# Modelos de Memória

// initially  $X = 1$  and  $X = 1$

Process P1:

$X \leftarrow 0;$

...

if ( $Y = 0$ ) then kill P2;

Process P2:

$Y \leftarrow 0;$

...

if ( $X = 0$ ) then kill P1;

Qual é a intenção da programadora?

No máximo um dos dois processos devem ser eliminados.

Poderiam os dois serem eliminados?

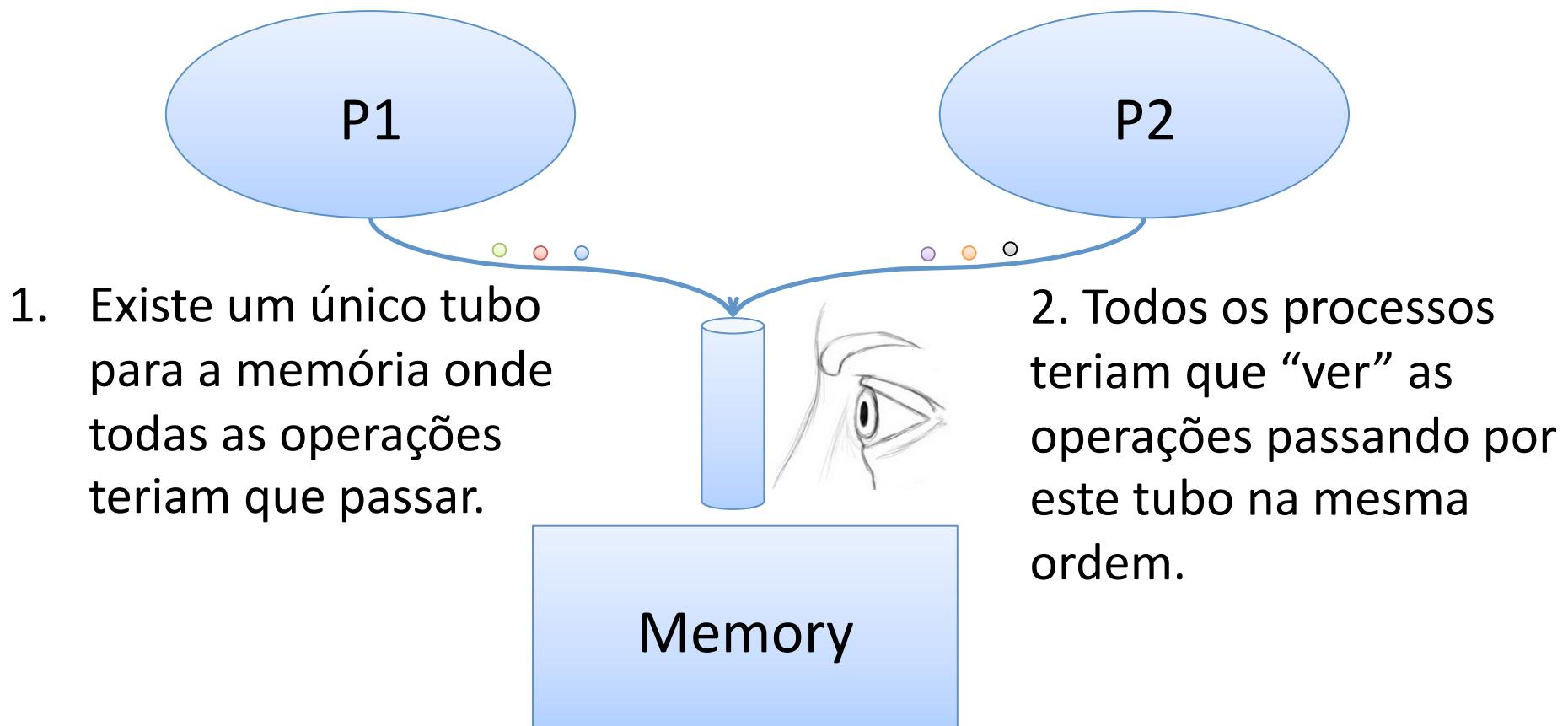
Se leituras podem pular sobre escritas os dois podem ser eliminados.

# Location Consistency

1. Cada processo é executado em alguma ordem sequencial que segue sua “ordem de programa.”
2. Todos os processos vêem a mesma ordem de operações

# Location Consistency

O Resultado é o mesmo como se fosse:



# Como garantir Location Consistency

- Garantindo *Program order*

*Program order* requer que cada instrução complete sua execução antes que a próxima instrução seja executada.

*Program order* proíbe reordenação de loads e stores.

- Program order resulta em *Sequential Consistency*

# Como implementar Sequential Consistency?

No write buffers?

(loads pulam sobre stores do ponto de vista de outros processadores)

É necessário aguardar por confirmação de invalidação/update de todas as caches antes de começar a execução da próxima instrução?

Loads não podem passar sobre stores?

Proibir loads especulativos?

# Sequential Consistency é um Modelo de Memória *Forte*

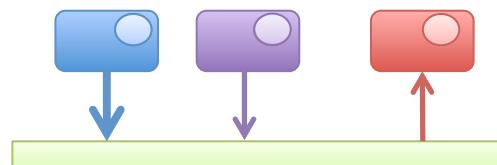
Ele requer que todas as operações de memória  
sejam *totalmente ordenadas* na ordem do  
programa.

É portanto muito restritivo resultando em perda de  
desempenho

# Desempenho de Sequential Consistency

- Primeira tentativa de melhorar desempenho

Configuração



multiprocessor  
com protocolo de coerência  
baseado em invalidação

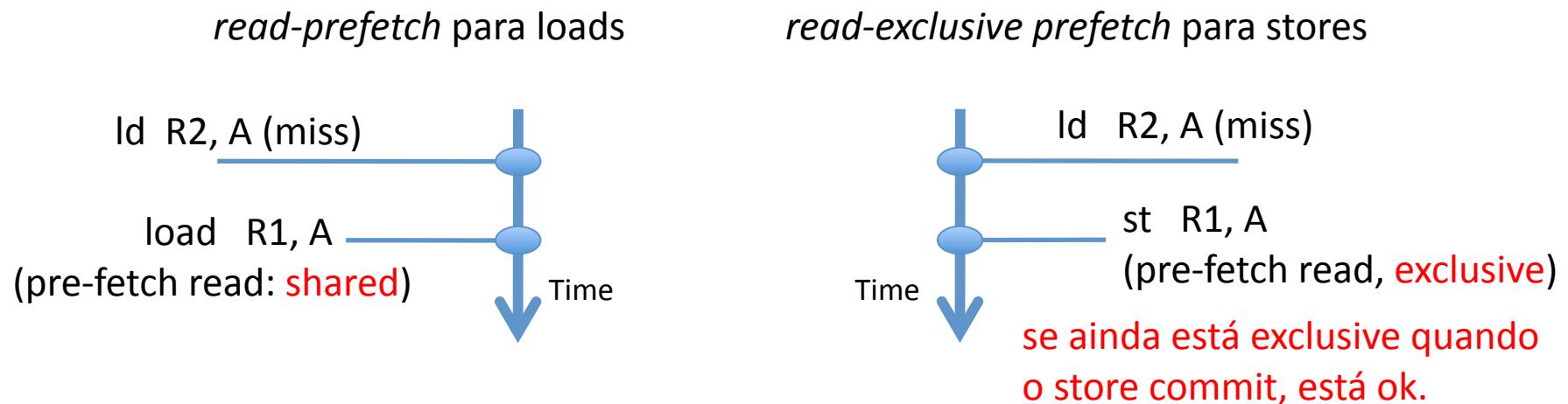
Processadores fora de ordem



protocolo de coerência  
baseado em snooping

# Pre-fetch

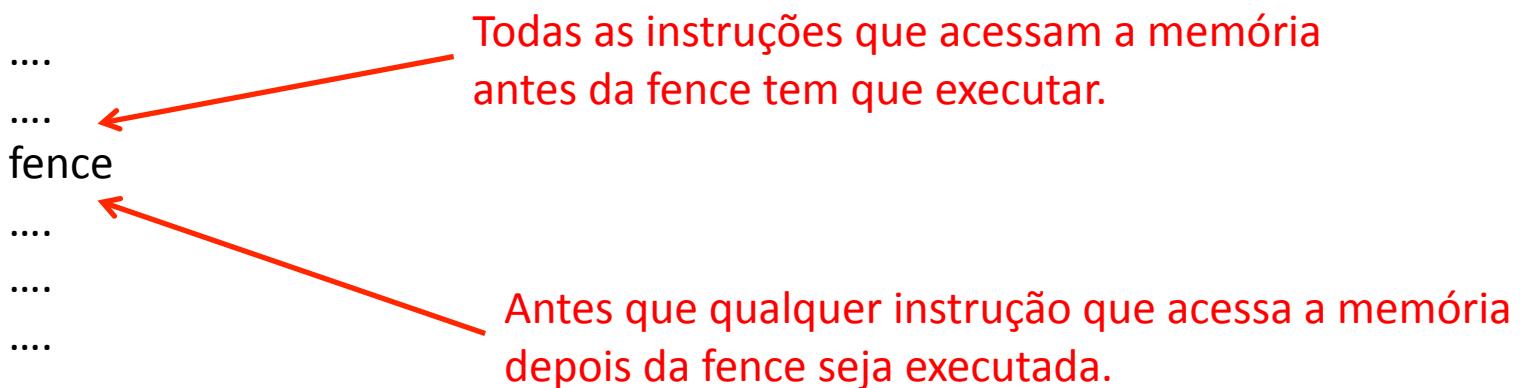
- Objetivo: reduzir os custos de cache misses
- Compensa espera por misses



- Especula que a posição de memória do load não será invalidada e continua execução
  - Verifica quando o load está no topo do reorder buffer.

# Modelos de Memória Relaxados

- *weak ordering*: alguns loads e stores são marcados como pontos de sincronização.
- Exemplo: uma instrução *fence*



# Processor Consistency

1. Escritas (stores) tem que ser executadas em program order
2. Leituras (loads) podem pular sobre stores
  - Uso de FIFO write buffers é ok.

# Exemplo de Processor Consistency

```
// initially flag = 0
```

Process P1:

```
write (A);  
flag ← 1
```

Process P2:

```
while (flag != 1); /* spin on flag */  
read (A);
```

Qual é a intenção do programador?

P1 produz o valor de A a ser consumido por P2

A Processor Consistency é equivalente à Sequential Consistency?

Sim, escritas para flag e A não podem ser reordenadas.

Stores não podem poder pular sobre stores

# Exemplo de Processor Consistency

// initially  $X = 1$  and  $X = 1$

Process P1:

$X \leftarrow 0;$

...

if ( $Y = 0$ ) then kill P2;

Process P2:

$Y \leftarrow 0;$

...

if ( $X = 0$ ) then kill P1;

Qual é a intenção da programadora?

No máximo um dos dois processos pode ser eliminados

Poderiam os dois ser eliminados?

Sim, em P1 a leitura de Y pode ocorrer antes da escrita de X  
e em P2 a leitura de X pode ocorrer antes da escrita de Y.

# Exemplo de Processor Consistency

// initially  $x = y = 0$

Process P1:

$R1 \leftarrow x$   
 $y \leftarrow 1$

Process P2:

$R2 \leftarrow y$   
 $x \leftarrow 1$

O resultado poderia ser  $R1 = R2 = 1$ ?

Não, stores não podem ser reordenados com loads mais antigos.

# Exemplo de Processor Consistency

// initially  $x = y = 0$

Process P1:

$x \leftarrow 1$

$R1 \leftarrow y$

Process P2:

$y \leftarrow 1$

$R2 \leftarrow x$

O resultado poderia ser  $R1 = R2 = 0$ ?

Sim, loads podem ser reordenados com stores mais velhos.

Se esta não era a intenção do programador então compilador e programador precisam estar cientes



# Regras para Processor Consistency

- Loads não são reordenados com outros loads
- Stores não são reordenados com outros stores
- Stores não são reordenados com loads mais velhos
- Em um multiprocessador stores para a mesma posição de memória tem uma ordem total
- Loads podem ser reordenados com stores mais antigos para posições diferentes de memória, mas não com stores para a mesma posição.

# Desempenho de Processor Consistency

Desempenho melhora por aproximadamente 10% em relação a sequential consistency para computações científicas.