计算机体系结构

周学海

xhzhou@ustc.edu.cn

0551-63601556, 63492271

中国科学技术大学

第7章　多处理器及线程级并行

- 7.1 引言
- 7.2 集中式共享存储器体系结构
- 7.3 分布式共享存储器体系结构
- 7.4 存储同一性
- 7.5 同步与通信

7.1、引言

- 单处理机的发展正在走向尽头？
- 并行计算机在未来将会发挥更大的作用。

1. 获得超过单处理器的性能，最直接的方法就是把多个处理器连在一起；

2. 自1985年以来，体系结构的改进使性能迅速提高，这种改进的速度能否持续下去还不清楚，但通过复杂度和硅技术的提高而得到的性能的提高正在减小；

3. 并行计算机应用软件已有缓慢但稳定的发展。

并行计算机体系结构的分类

1、按照Flynn分类法，可把计算机分成

- 单指令流单数据流（SISD）
- 单指令流多数据流（SIMD）
- 多指令流单数据流（MISD）
- 多指令流多数据流（MIMD）

2、MIMD已成为通用多处理机体系结构的选择，原因：

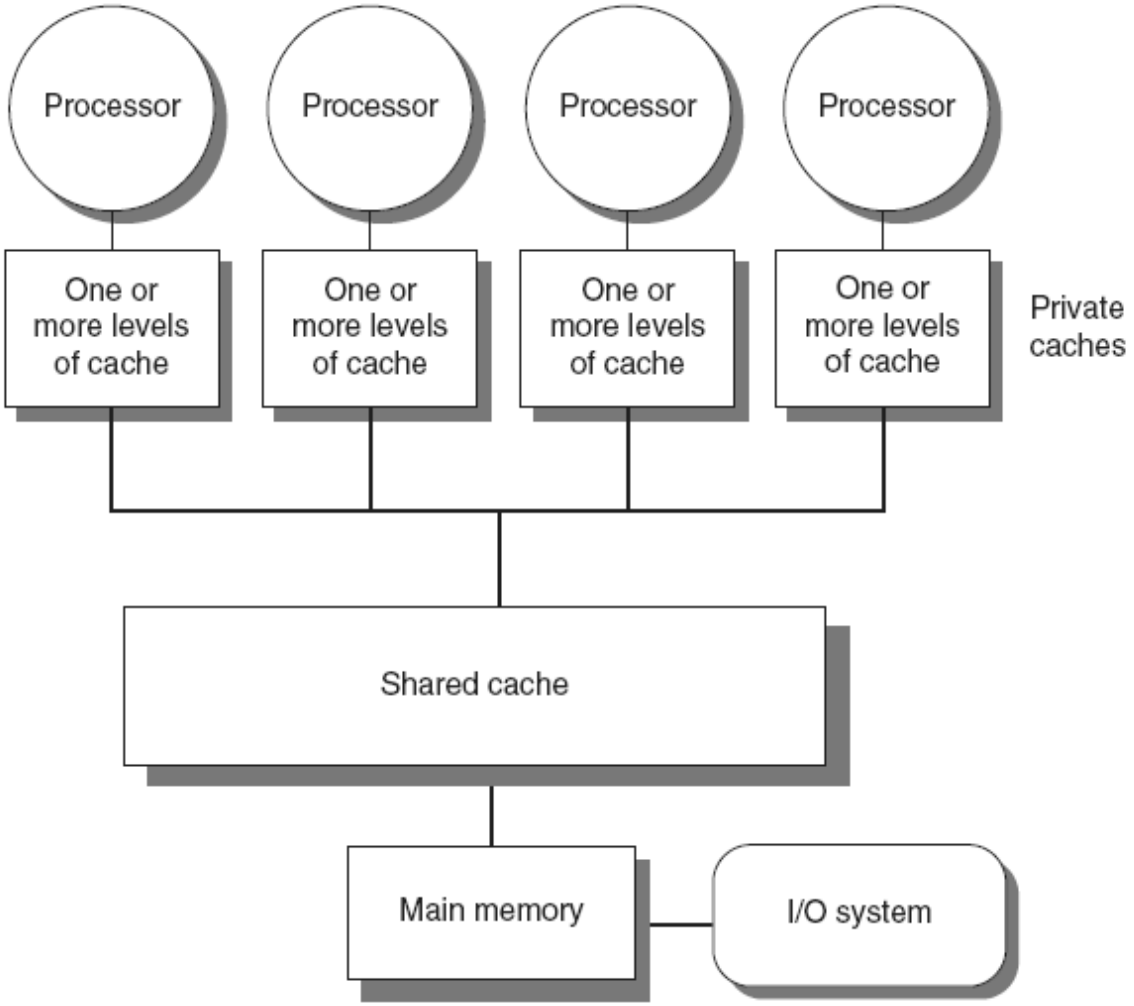
(1) MIMD具有灵活性。

(2) MIMD可以充分利用商品化微处理器在性能价格比方面的优势。

3、根据多处理机系统中处理器个数的多少，可把现有的MIMD机器分为两类（每一类代表了一种存储器的结构和互连策略）

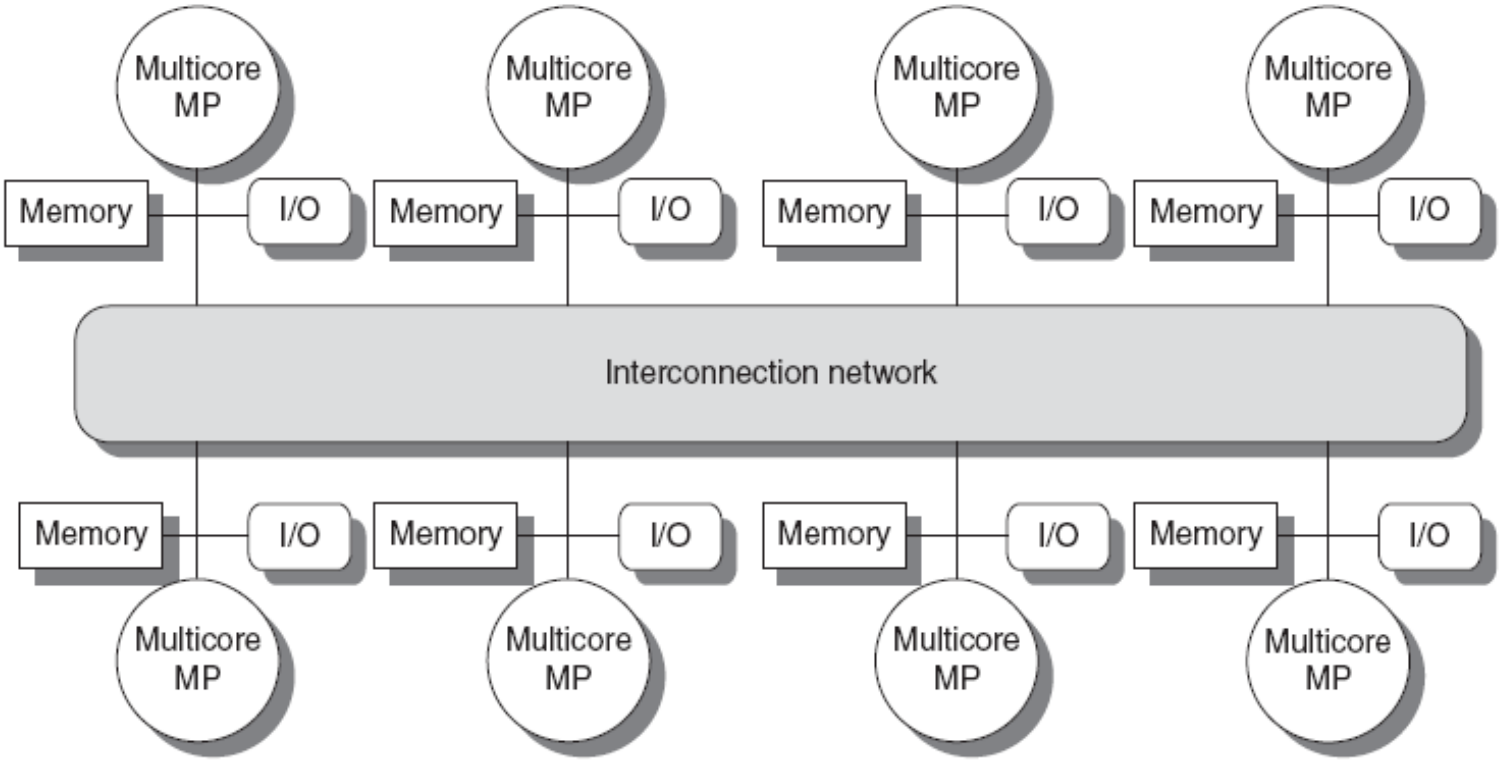
(1) 集中式共享存储器结构

这类机器有时被称为UMA(uniform memory access)机器。


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

(2) 分布式存储器结构

- 每个结点包含：
  - 处理器
  - 存储器
  - I/O
- 在许多情况下，分布式存储器结构优于采用集中式共享存储器结构。
- 分布式存储器结构需要高带宽的互连。


Multicore MP
Multicore MP
Multicore MP
Multicore MP
Memory
I/O
Memory
I/O
Memory
I/O
Memory
I/O
Interconnection network
Memory
I/O
Memory
I/O
Memory
I/O
Memory
I/O
Multicore MP
Multicore MP
Multicore MP
Multicore MP

- 分布式存储器结构的优点
  - ✓如果大多数的访问是针对本结点的局部存储器，则可降低对存储器和互连网络的带宽要求；
  - ✓对局部存储器的访问延迟低。
- 主要缺点

  处理器之间的通信较为复杂，且各处理器之间访问延迟较大。
- 簇：超结点

通信模型和存储器的结构模型

1. 两种地址空间的组织方案

(1)物理上分离的多个存储器可作为一个逻辑上共享的存储空间进行编址。这类机器的结构被称为分布式共享存储器(DSM)或可缩放共享存储器体系结构。DSM机器被称为NUMA(non-uniform memory access)机器。

(2)整个地址空间由多个独立的地址空间构成，它们在逻辑上也是独立的，远程的处理器不能对其直接寻址。

两种通信模型

每一个处理器-存储器模块实际上是一个单独的计算机，这种机器也称为多计算机。

2. 两种通信模型

(1) 共享地址空间的机器

- ✓利用Load和Store指令中的地址隐含地进行数据通讯。

(2)多个地址空间的机器

- ✓通过处理器间显式地传递消息完成
- ✓这种机器常称为消息传递机器。

通信模式

消息传递机器根据简单的网络协议，通过传递消息来请求某些服务或传输数据，从而完成通信。

例如:一个处理器要对远程存储器上的数据进行访问或操作

① 发送消息，请求传递数据或对数据进行操作.远程进程调用(RPC,remote process call)

② 目的处理器接收到消息以后，执行相应的操作或代替远程处理器进行访问，并发送一个应答消息将结果返回。

通信模式

同步消息传递

- ✓请求处理器发送一个请求后一直要等到应答结果才继续运行。

异步消息传递

- ✓发送方发出消息后，立即继续执行原来的程序。
- ✓发送方不先经请求就直接把数据送往数据接受方。

通信机制的性能

三个关键的性能指标：

① 通信带宽

理想状态下的通信带宽受限于处理器、存储器和互连网络的带宽。

② 通信延迟

理想状态下通信延迟应尽可能地小。

通信延迟＝发送开销＋跨越时间＋传输延迟
＋接收开销

③ 通讯延迟的隐藏

✓如何才能较好地将通信和计算或多次通信之间重叠起来，以实现通讯延迟的隐藏。

✓通常的原则是：只要可能就隐藏延迟。

✓通信延迟隐藏是一种提高性能的有效途径，但它对操作系统和编程者来讲增加了额外的负担。

不同通信机制的优点

1、共享存储器通信的主要优点

① 与常用的集中式多处理机使用的通信机制兼容

② 当处理器通信方式复杂或程序执行动态变化时易于编程，同时在简化编译器设计方面也占有优势。

③ 当通信数据较小时，通信开销较低，带宽利用较好。

④ 通过硬件控制的Cache减少了远程通信的频度，减少了通信延迟以及对共享数据的访问冲突。

2. 消息传递通信机制的主要优点

① 硬件较简单。

② 通信是显式的，从而引起编程者和编译程序的注意，着重处理开销大的通信。

- 在共享存储器上支持消息传递相对简单，但在消息传递的硬件上支持共享存储器就困难得多。
- 所有对共享存储器的访问均要求操作系统提供地址转换和存储保护功能，即将存储器访问转换为消息的发送和接收。

并行处理面临的挑战

并行处理面临着两个重要的挑战：

- 程序中有限的并行性
- 相对较高的通信开销

1、有限的并行性使机器要达到好的加速比十分困难

例7.1 如果想用100个处理器达到80的加速比，求原计算程序中串行部分所占比例。

例7.1

解　　根据Amdahl定律加速比为：

$$\frac{1}{\frac{\text{可加速部分比例}}{\text{理论加速比}} + (1-\text{可加速部分比例})}$$

$$80 = \frac{1}{\frac{\text{并行比例}}{100} + (1-\text{并行比例})}$$

- 得出：并行比例＝0.9975
- 　　可以看出要用100个处理器达到80的加速比，串行计算的部分只能占0.25%。

2. 面临的第二个挑战主要是指多处理机中远程访问的较大延迟。

在现有的机器中，处理器之间的数据通信大约需要35～>500个时钟周期。

同一芯片中core之间的延迟35~50cycles

不同芯片间core之间的延迟100~>500 cycles

远程访问一个字的延迟时间

机　器	通信机制	互连网络	处理机数量	典型远程存储器访问时间
SPARC Center	共享存储器	总线	≤20	1μs
SGI Challenge	共享存储器	总线	≤36	1μs
Cray T3D	共享存储器	3维环网	32－2048	1μs
Convex Exemplar	共享存储器	交叉开关＋环	8－64	2μs
KSR-1	共享存储器	多层次环	32－256	2-6μs
CM-5	消息传递	胖树	32－1024	10μs
Intel Paragon	消息传递	2维网格	32－2048	10-30μs
IBM SP-2	消息传递	多级开关	2－512	30-100μs

例7.2　一台32个处理器的计算机，对远程存储器访问时间为2000ns。除了通信以外，假设计算中的访问均命中局部存储器。当发出一个远程请求时，本处理器挂起。处理器时钟时间为10ns，如果指令基本的CPI为1.0(设所有访存均命中Cache)，求在没有远程访问的状态下与有0.5%的指令需要远程访问的状态下，前者比后者快多少?

解　有0.5%远程访问的机器的实际CPI为

CPI＝基本CPI＋远程访问率×远程访问开销

　　＝1.0＋0.5%×远程访问开销

远程访问开销＝远程访问时间/时钟时间

　　＝2000ns/10ns＝200个时钟

∴　CPI＝1.0＋0.5%×200＝2.0

它为只有局部访问的机器的2.0 / 1.0＝2倍，

因此在没有远程访问的状态下的机器速度是有0.5%远程访问的机器速度的2倍。

问题的解决

- 并行性不足：
  - ✓通过采用并行性更好的算法来解决
- 远程访问延迟的降低：
  - ✓靠体系结构支持和编程技术

并行程序的计算 / 通信比率

- 反映并行程序性能的一个重要的度量

  计算与通信的比率

- 计算/通信比率随着处理数据规模的增大而增加；随着处理器数目的增加而降低。

7.2 集中式共享存储器体系结构

多个处理器共享一个存储器。

- 当处理器规模较小时，这种机器十分经济。
- 支持对共享数据和私有数据的Cache缓存。

私有数据供一个单独的处理器使用，而共享数据供多个处理器使用。

- 共享数据进入Cache产生了一个新的问题

Cache的一致性问题

多处理机的一致性

(1) 不一致产生的原因（Cache一致性问题）

- I／O操作
  - ✓Cache中的内容可能与由I／O子系统输入输出形成的存储器对应部分的内容不同。
- 共享数据
  - ✓不同处理器的Cache都保存有对应存储器单元的内容。

例　两个处理器Cache对应同一存储器单元产生出不同的值

假设：初始条件下各个Cache无X值，X单元值为1；写直达方式的Cache。

时间　事件	CPU A Cache 内容	CPU B Cache 内容	X单元存储器内容
0			1
1 CPU A读X	1		1
2 CPU B读X		1	1
3 CPU A将0存入X	0	1	0

（2）存储器是一致的（非正式地定义）

如果对某个数据项的任何读操作均可得到其最新写入的值，则认为这个存储系统是一致的。

- 存储系统行为的两个不同方面
  - ✓返回给读操作的是什么值(coherence)
  - ✓什么时候才能将已写入的值返回给读操作(consistency)
- 如果存储系统行为满足条件

① 处理器P对X进行一次写之后又对X进行读，读和写之间没有其它处理器对X进行写，则读的返回值总是写进的值。

② 一个处理器对X进行写之后，另一处理器对X进行读，读和写之间无其它写，则读X的返回值应为写进的值。

③ 对同一单元的写是顺序化的，即任意两个处理器对同一单元的两次写，从所有处理器看来顺序都应是相同的。

□2点假设

（1）直到所有的处理器均看到了写的结果，一次写操作才算完成；

（2）允许处理器无序读，但必须以程序规定的顺序进行写。

实现一致性的基本方案

在一致的多处理机中，Cache提供两种功能。

- 共享数据的迁移
  - ✓降低了对远程共享数据的访问延迟。
- 共享数据的复制
  - ✓不仅降低了访存的延迟，也减少了访问共享数据所产生的冲突。

小规模多处理机不是采用软件而是采用硬件技术实现Cache一致性。

(1)Cache一致性协议:

对多个处理器维护一致性的协议

(2) 关键：跟踪共享数据块的状态

(3) 共享数据状态跟踪技术

- 目录
  - ✓物理存储器中共享数据块的状态及相关信息均被保存在一个称为目录的地方。
- 监听
  - ✓每个Cache除了包含物理存储器中块的数据拷贝之外，也保存着各个块的共享状态信息。

Cache通常连在共享存储器的总线上，各个Cache控制器通过监听总线来判断它们是否有总线上请求的数据块。

3. 两种协议

(1) 写作废协议

在一个处理器写某个数据项之前保证它对该数据项有唯一的访问权。

例 在写回Cache的条件下，监听总线中写作废协议的实现。

处理器行为	总线行为	CPU A Cache内容	CPU B Cache内容	主存X单元内容
				0
CPU A 读X	Cache失效	0		0
CPU B 读X	Cache失效	0	0	0
CPUA将X单元写1	作废X单元	1		0
CPU B 读X	Cache失效	1	1	1

(2) 写更新协议

当一个处理器写某数据项时，通过广播使其它Cache中所有对应的该数据项拷贝进行更新。

例　在写回Cache的条件下，监听总线中写更新协议的实现。

处理器行为	总线行为	CPUA Cache内容	CPUB Cache内容	主存X单元内容
				0
CPU A 读X	Cach失效	0		0
CPU B 读X	Cach失效	0	0	0
CPUA将X单元写1	广播写X单元	1	1	1
CPU B 读X		1	1	1

(3) 写更新和写作废协议性能上的差别

- 对同一数据的多个写而中间无读操作的情况,写更新协议需进行多次写广播操作，而在写作废协议下只需一次作废操作。
- 对同一块中多个字进行写，写更新协议对每个字的写均要进行一次广播，而在写作废协议下仅在对本块第一次写时进行作废操作。
- 一个处理器写到另一个处理器读之间的延迟通常在写更新模式中较低。而在写作废协议中，需要读一个新的拷贝。

在基于总线的多处理机中，写作废协议成为绝大多数系统设计的选择。

4. 监听协议的基本实现技术

(1) 小规模多处理机中实现写作废协议的关键利用总线进行作废操作,每个块的有效位使作废机制的实现较为容易。

(2) 写直达Cache，因为所有写的数据同时被写回主存，则从主存中总可以取到最新的数据值。

(3) 对于写回Cache，得到数据的最新值会困难一些，因为最新值可能在某个Cache中，也可能在主存中。

(4)在写回Cache条件下的实现技术

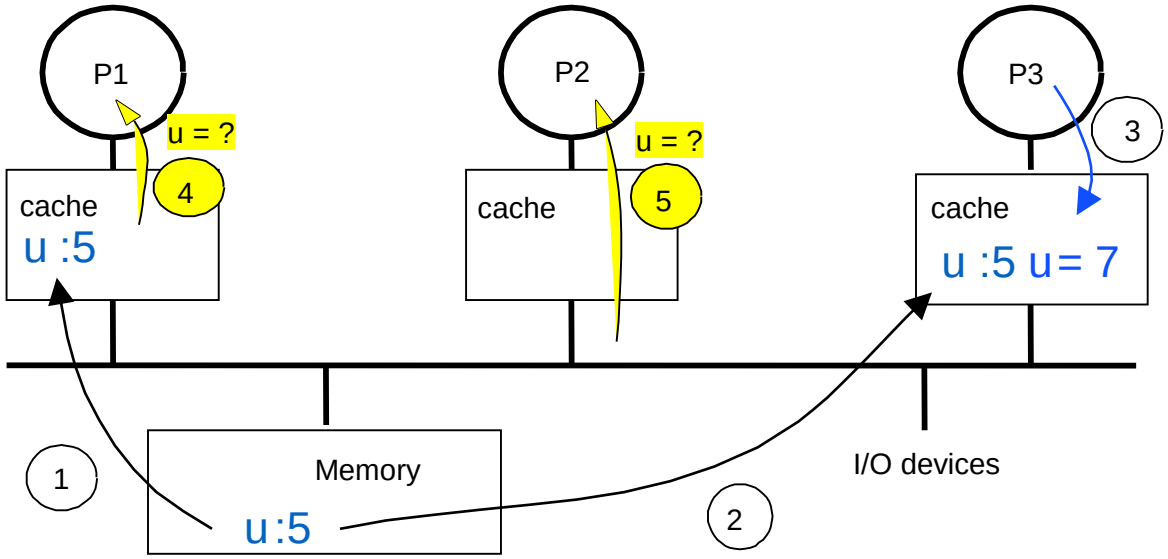
- ✓用Cache中块的标志位实现监听过程。
- ✓给每个Cache块加一个特殊的状态位说明它是否为共享。

Cache块的拥有者：拥有唯一的Cache块拷贝的处理器。

- ✓因为每次总线任务均要检查Cache的地址位，这可能与CPU对Cache的访问冲突。可通过下列两种技术之一降低冲突：

  复制标志位或采用多级包含Cache。

Example on Cache Coherence Problem



P3执行了写操作后，处理器看到了u的不同值

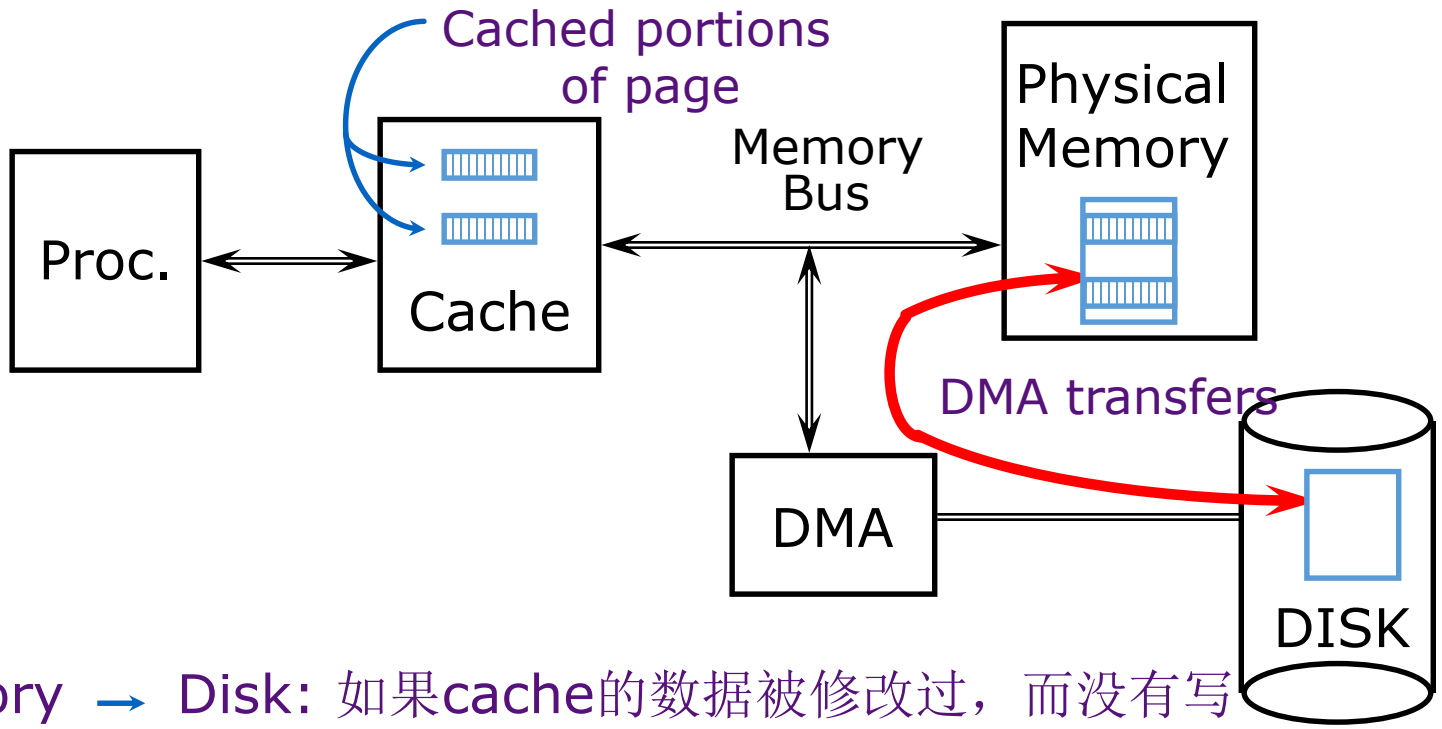
针对write back caches …

处理器访问主存可能看到一个陈旧的（不正确）值

结果写依赖于cache Flush的顺序

显然，这样对程序的执行是不可接受的

Problems with Parallel I/O

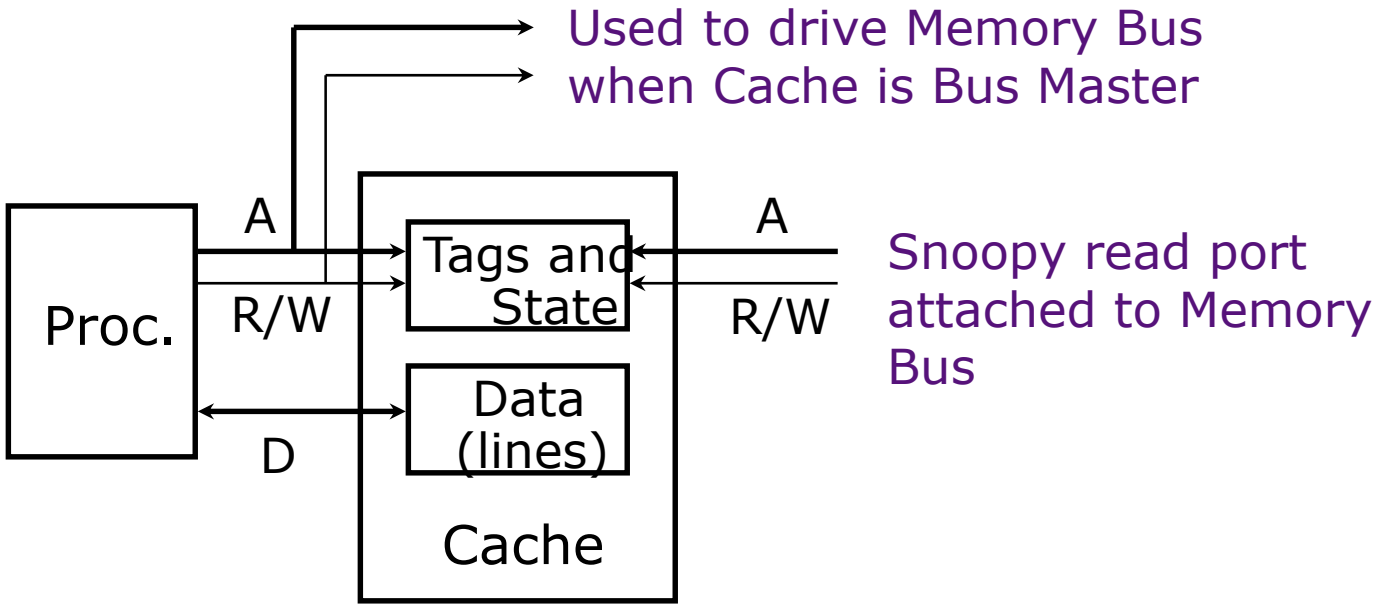


Memory → Disk: 如果cache的数据被修改过，而没有写回，存储器是陈旧数据

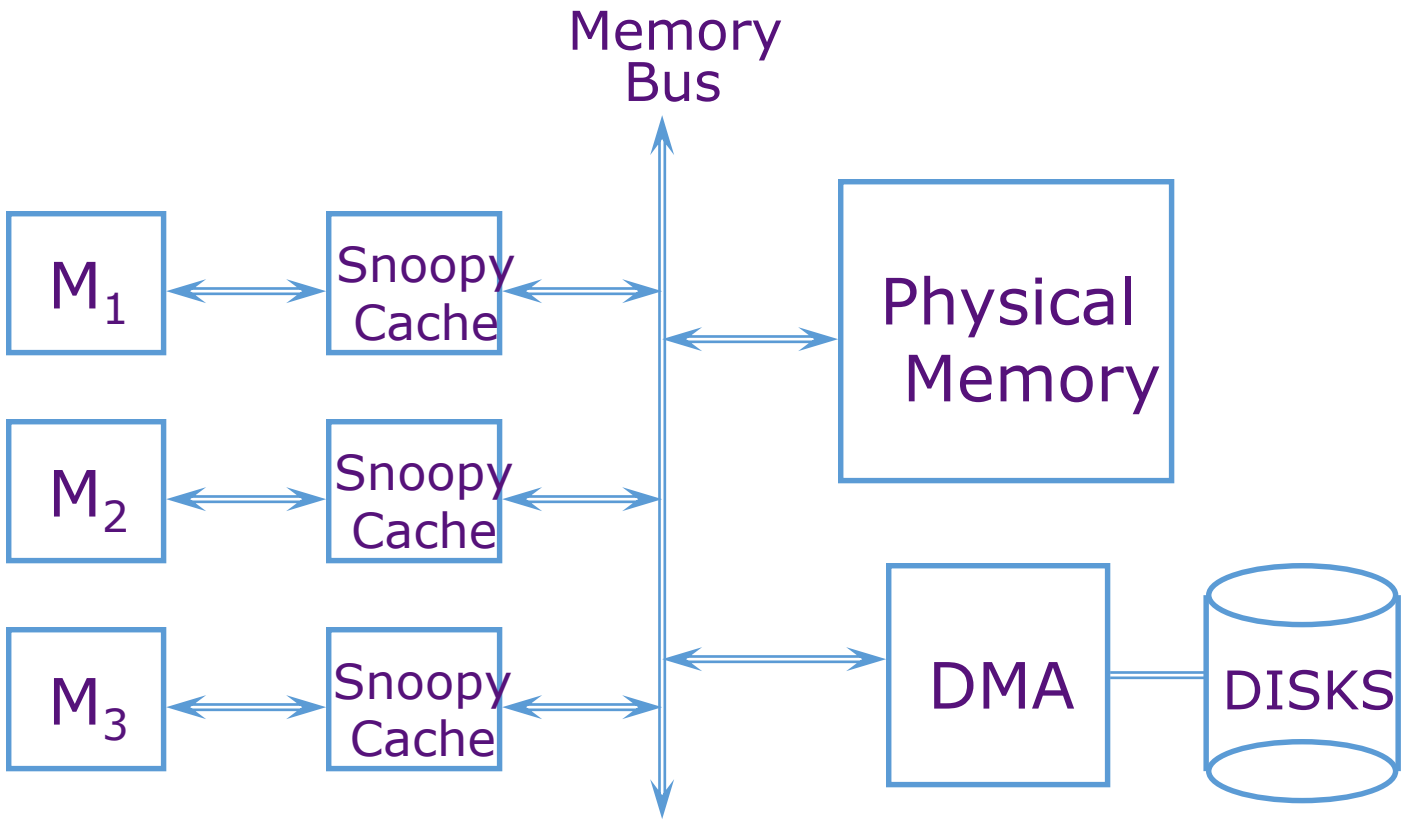
Disk → Memory: Cache中的数据是陈旧数据，它并不知道这次存储器写操作

Snoopy Cache, *Goodman 1983*

- Idea: 让Cache监测 DMA 传输, 然后 “do the right thing”
- 双端口的Snoopy cache tags

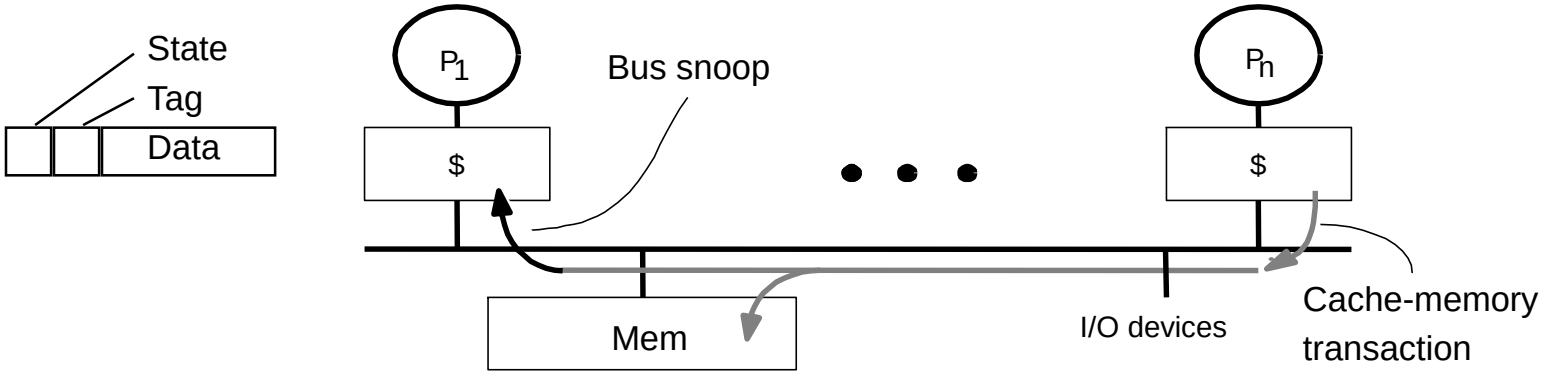


Shared Memory Multiprocessor



利用监听机制来保持处理器看到的存储器的视图一致

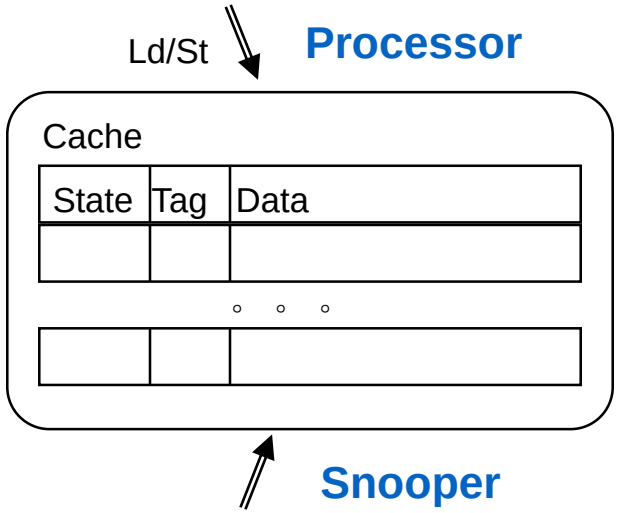
Snoopy Cache-Coherence Protocols



- Bus is a broadcast medium & caches know what they have
  - 总线上的事务对所有Cache是可见的
- Cache 控制器监测 （**snoop**）**共享总线上的所有事务**
  - 根据Cache中的块其状态不同会产生不同的事务
  - 通过执行不同的总线事务来保证Cache的一致性
    - Invalidate, update, or supply value

Implementing a Snooping Protocol

- Cache 控制器接收两方面的请求输入：
  - 处理器的请求 (load/store)
  - 监测器（snooper)的总线请求/响应
- 控制器根据这两方面的输入产生相应的动作
  - 更新Cache块的状态
  - 提供数据
  - 产生新的总线事务



Snoopy Cache Coherence Protocols

write miss:

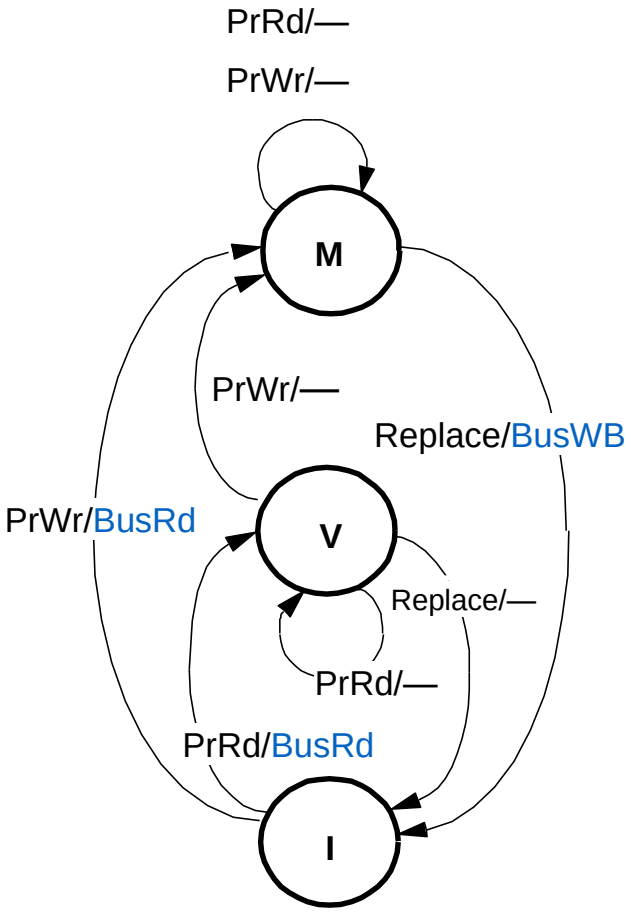
写操作完成前，其他cache中包含该地址的块作废（invalidated）

read miss:

如果有些Cache中有包含该地址的脏块（dirty），在读操作完成前将脏块写回

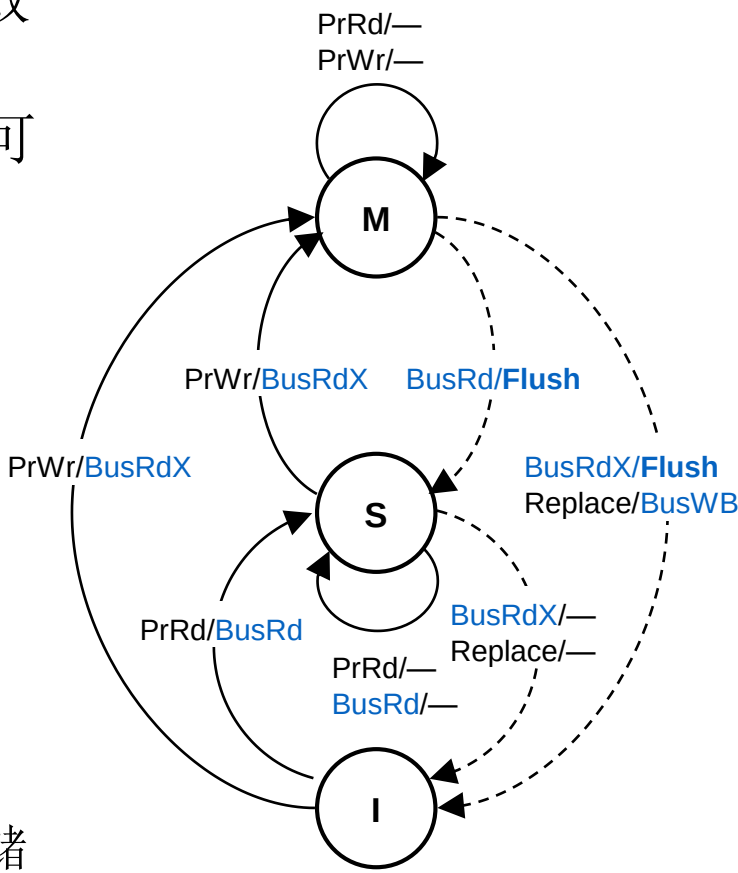
Write-back Cache

- Processor / Cache 操作
  - **PrRd**, **PrWr**, block **Replace**
- Cache块状态
  - **Invalid**, **Valid** (clean), **Modified** (dirty)
- 总线事务
  - Bus Read (**BusRd**), Write-Back (**BusWB**)
  - 仅传送cache-block
- 针对Cache一致性的块状态调整
  - Treat Valid as **Shared**
  - Treat Modified as **Exclusive**
- 引入新的总线事务
  - **Bus Read-eXclusive (BusRdX)**
  - 其基本动作是：读进并修改



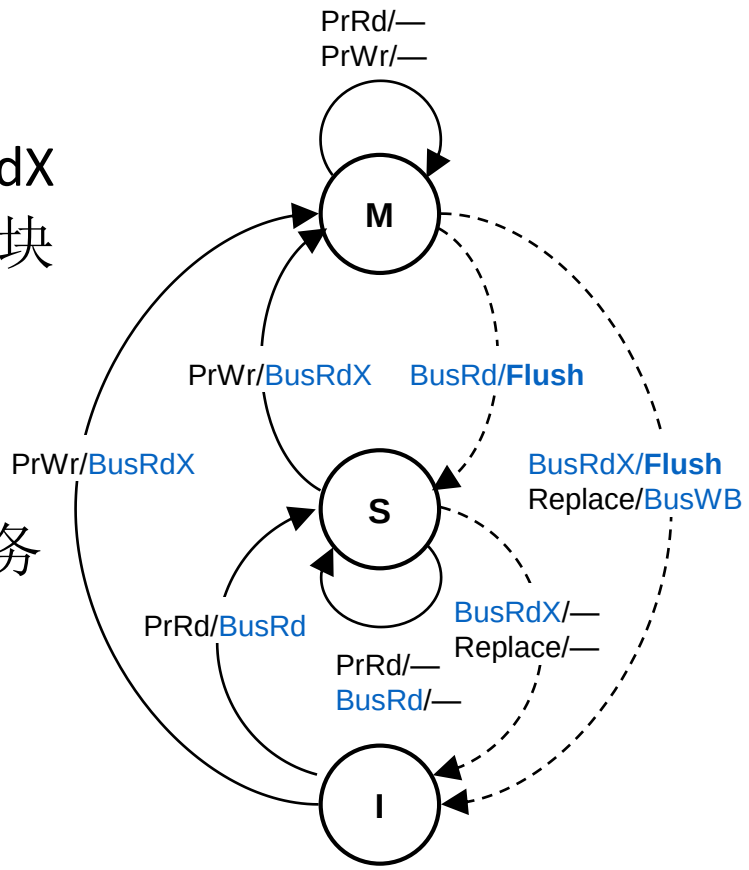
MSI Write-Back Invalidate Protocol

- 3 states:
  - **Modified**: 仅该cache拥有修改过的、有效的该块copy
  - **Shared**: 该块是干净块，其他cache中也可能含有该块，存储器中的内容是最新的
  - **Invalid**: 该块是无效块（invalid）
- 4 bus transactions:
  - Bus Read: **BusRd** on a read miss
  - Bus Read Exclusive: **BusRdX**
    - 得到独占的（exclusive）cache block
    - 其基本动作为读进并修改
  - Bus Write-Back: **BusWB** on replacement
  - **Flush** on BusRd or BusRdX
    - Cache将数据块放到总线上（而不是从存储器取数据）完成 Cache-to-cache的传送，并更新存储器

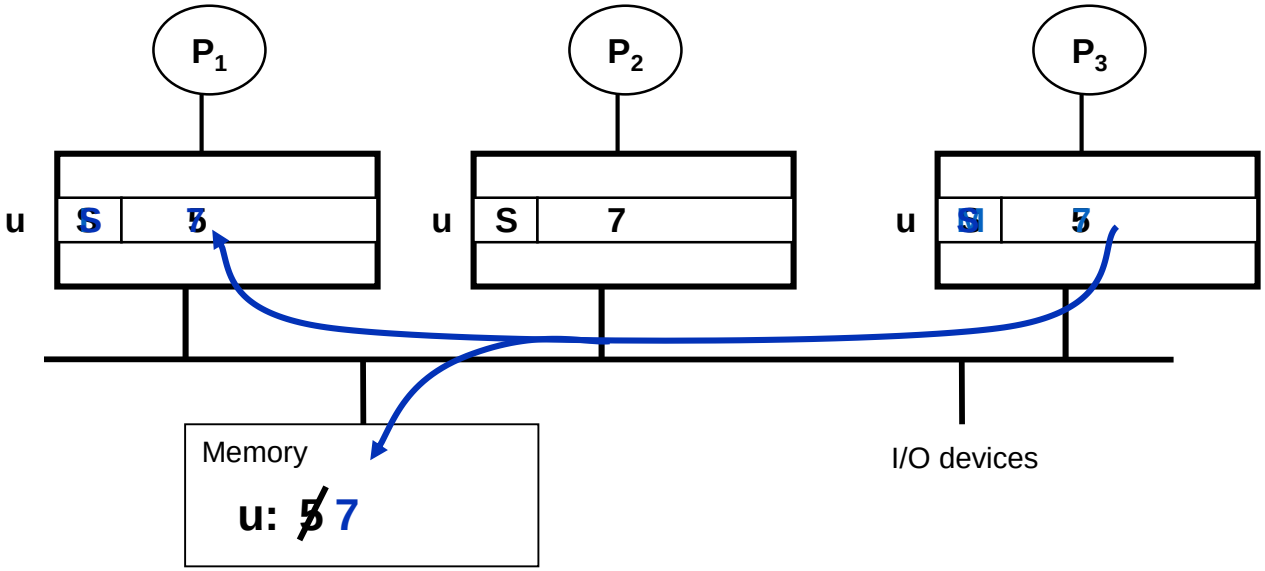


State Transitions in the MSI Protocol

- Processor Read
  - Cache miss ⇒ 产生BusRd事务
  - Cache hit (S or M) ⇒ 无总线动作
- Processor Write
  - 当在非Modified状态时，产生总线BusRdX事务，BusRdX 导致其他Cache中的对应块作废（invalidate）
  - 当在Modified状态时，无总线动作
- Observing a Bus Read
  - 如果该块是 Modified, 产生Flush总线事务
    - 更新存储器和有需求的Cache
    - 引起总线事务的Cache块状态⇒ Shared
- Observing a Bus Read Exclusive
  - 作废相关block
  - 如果该块是modified, 产生Flush总线事务



Example on MSI Write-Back Protocol



Processor Action	State P1	State P2	State P3	Bus Action	Data from
1. P1 reads u	S			BusRd	Memory
2. P3 reads u	S		S	BusRd	Memory
3. P3 writes u	I		M	BusRdX	Memory
4. P1 reads u	S		S	BusRd, Flush	P3 cache
5. P2 reads u	S	S	S	BusRd	Memory

Lower-level Design Choices

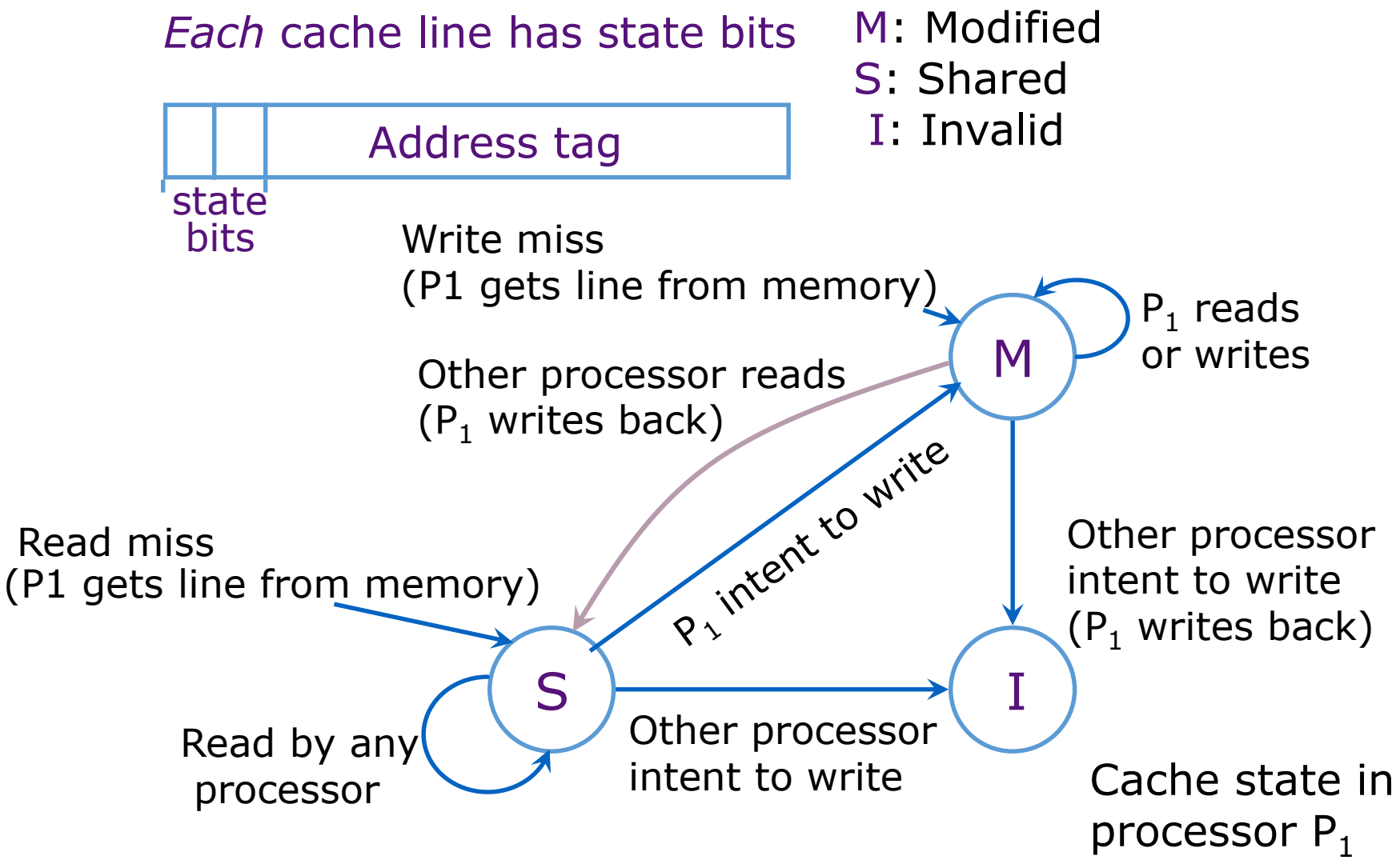
- Bus Upgrade (**BusUpgr**) 将Cache块的状态从 S 到M
  - 引起作废操作 (类似 BusRdX)，但避免块的读操作
- 当 BusRd 由M态的块引起时，变迁到哪个态
  - M → S or M → I 取决于访问模式
- Transition to state S
  - 如果不久会有本地读操作，而不是其他处理器的写操作
  - 比较适合于经常发生读操作的访问模式
- Transition to state I
  - 经常发生其他处理器写操作
  - 比较适合数据迁移操作：即本地写后，其他处理器将会发出读和写请求，然后本地又进行读和写。即连续的对称式访问模式
- 不同选择方案会影响存储器的性能

MSI Snooping Cache Coherence Protocol

Request	Source	State Transition	Action and Explanation
Read Hit	Processor	Shared or Modified	Normal Hit: Read data in private data cache (no transaction)
Read Miss	Processor	Invalid → Shared	Normal Miss: Place **read miss on bus**, change state
Read Miss	Processor	Shared	Replace block: Place **read miss on bus**
Read Miss	Processor	Modified → Shared	**Write-Back** block, Place **read miss on bus**, change state
Write Hit	Processor	Modified	Normal Hit: Write data in private data cache (no transaction)
Write Hit	Processor	Shared → Modified	Coherence: Place **invalidate on bus** (no data), change state
Write Miss	Processor	Invalid → Modified	Normal Miss: Place **write miss on bus**, change state
Write Miss	Processor	Shared → Modified	Replace block: Place **write miss on bus**, change state
Write Miss	Processor	Modified	**Write-Back** block, Place **write miss on bus**
Read Miss	Bus	Shared	**Serve read miss** from shared cache or memory
Read Miss	Bus	Modified → Shared	Coherence: **Write-Back & Serve read miss**, change state
Invalidate	Bus	Shared → Invalid	Coherence: **Invalidate shared block** in other private caches
Write Miss	Bus	Shared → Invalid	Coherence: **Invalidate shared block** in other private caches
Write Miss	Bus	Modified → Invalid	Coherence: **Write-Back & Serve write miss, Invalidate**

Cache State Transition Diagram

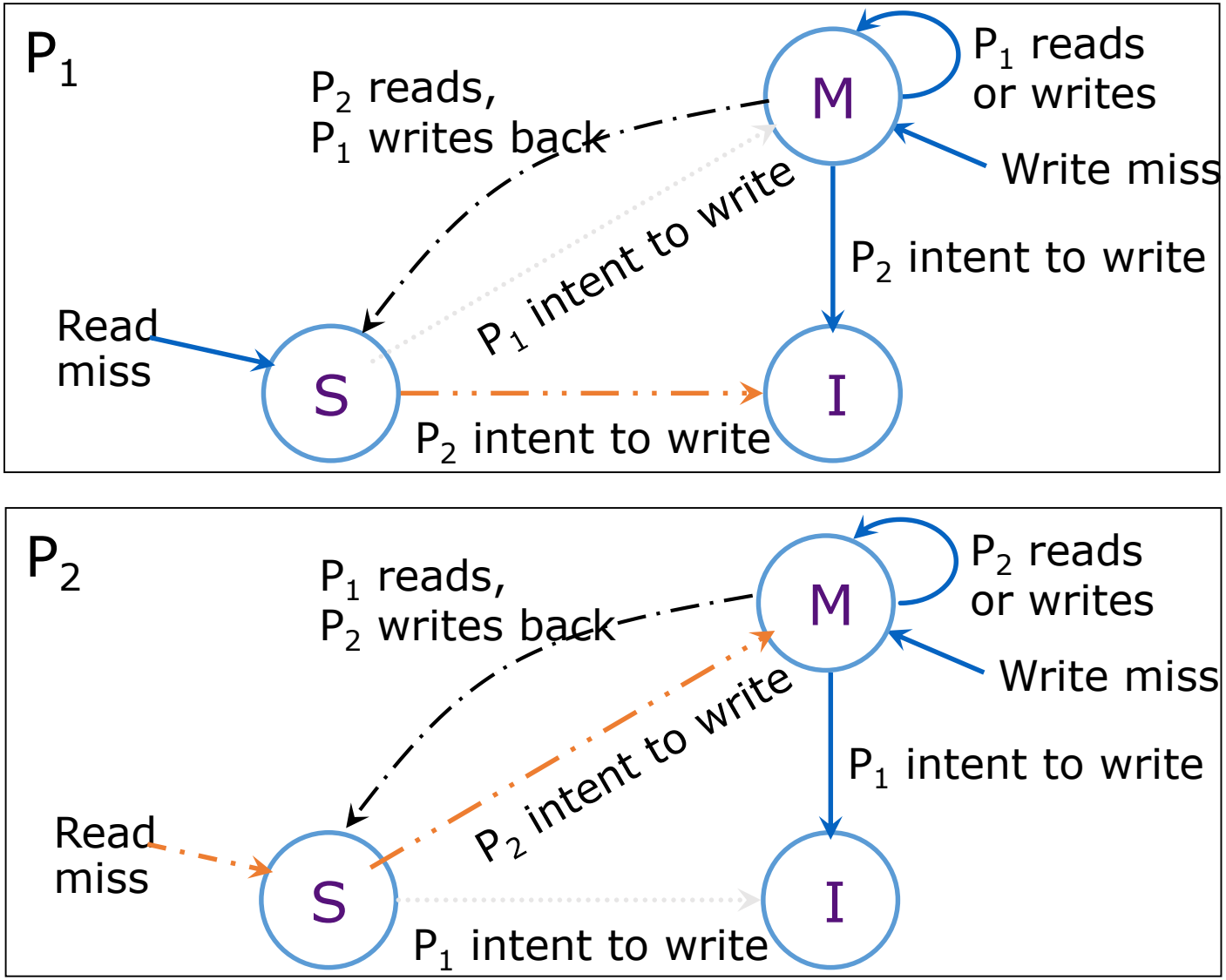
The MSI protocol



Two Processor Example

(Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes



Satisfying Coherence

- 写传播(Write propagation)
  - 对一个shared 或invalid块的写，其他cache都可见
    - 使用Bus Read-exclusive (BusRdX) 事务
    - Bus Read-exclusive 事务作废其他Cache中的块
    - 其他处理器在未看到该写操作的效果前体验到的是Cache Miss
- 写串行(Write serialization)
  - 所有出现在bus上的写操作(BusRdX)被总线串行化
    - 所有处理器（包括发出写操作的处理器）以同样的方式排序
    - 首先更新发出写操作的处理器的本地cache，然后处理其他事务
  - 但是并不是所有的写操作都会出现在总线上
    - 对modified 块的写序列来自同一个处理器， say *P*
    - 同一处理器是串行化的写：由P进行读操作将会看到串行序的写序列
    - 其他处理器的写串行化
      - 其他处理器的读操作会触发一个总线事务，将这个读的完成和写命中分开。
      - 保证这些写以相同的序呈现给其他处理器

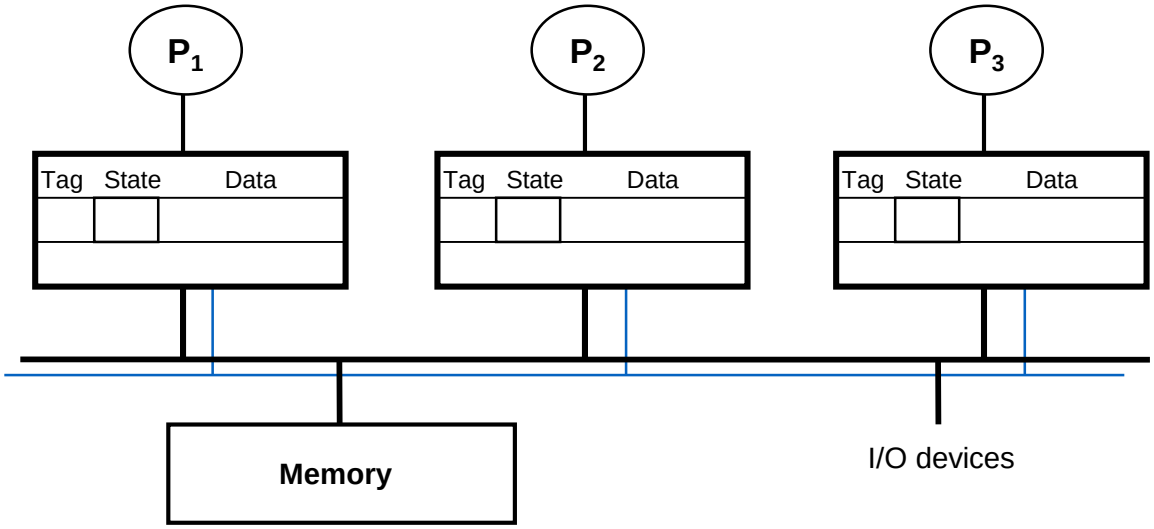
MESI Write-Back Invalidation Protocol

- MSI Protocol的缺陷：
  - 读进并修改一个block，产生2 个总线事务
    - 首先是读操作产生 BusRd (I→S)，并置状态为Shared, 写更新时产生 BusRdX (S→M)
    - 即时一个块是Cache独占的这种情况仍然存在
    - 使用多道程序负载时，这种情况很普遍
- 增加exclusive state, 减少总线事务
  - Exclusive state 表示仅当前包含该块，并且是干净的块
  - 区分独占块的“clean”和“dirty”
  - 一个处于exclusive state的块，更新时不产生总线事务

Four States: MESI

- **M: Modified**
  - 仅当前Cache含有该块，并且该块被修改过
  - 内存中的Copy是陈旧的值
- **E: Exclusive** or *exclusive-clean*
  - 仅当前Cache含有该块，并且该块没被修改过
  - 内存中的数据是最新的
- **S: Shared**
  - 多个Cache中都含有本块，而且都没有修改过
  - 内存中的数据是最新的
- **I: Invalid**
- 也称Illinois protocol
  - 首先是由的研究人员研制并发表论文
  - MESI协议的变种广泛应用于现代微处理器中

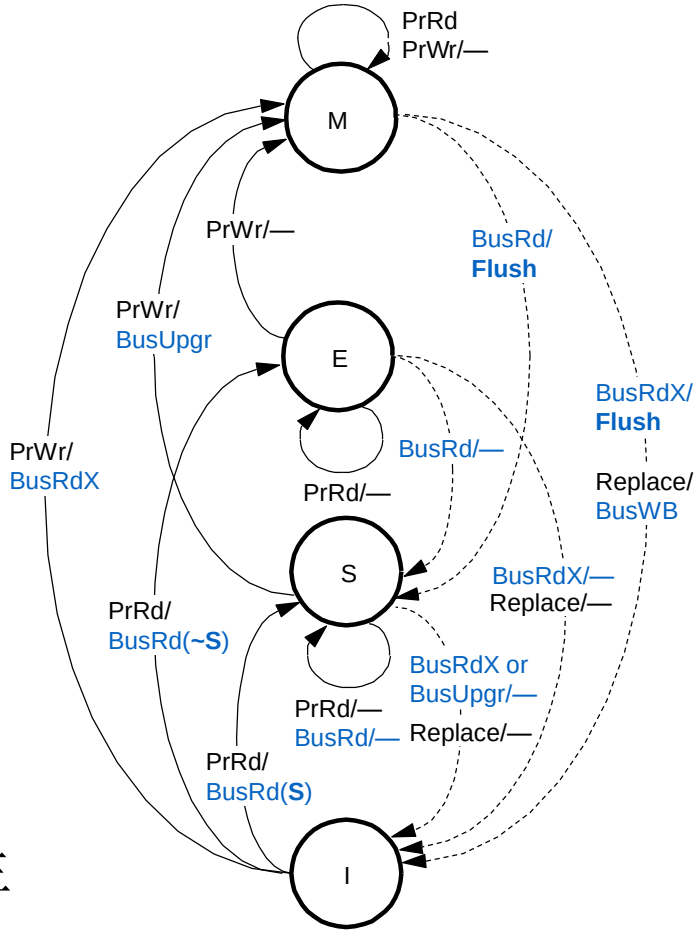
Hardware Support for MESI



- 总线互连的新要求
  - 增加一个称为shared signal *S*, 必须对所有Cache控制器可用
  - 可以实现成 wired-OR line
- 所有cache controllers 监测 BusRd
  - 如果所访问的块的状态是（state *S*, *E*, or *M*)
  - 请求Cache 根据shared signal选择E或S

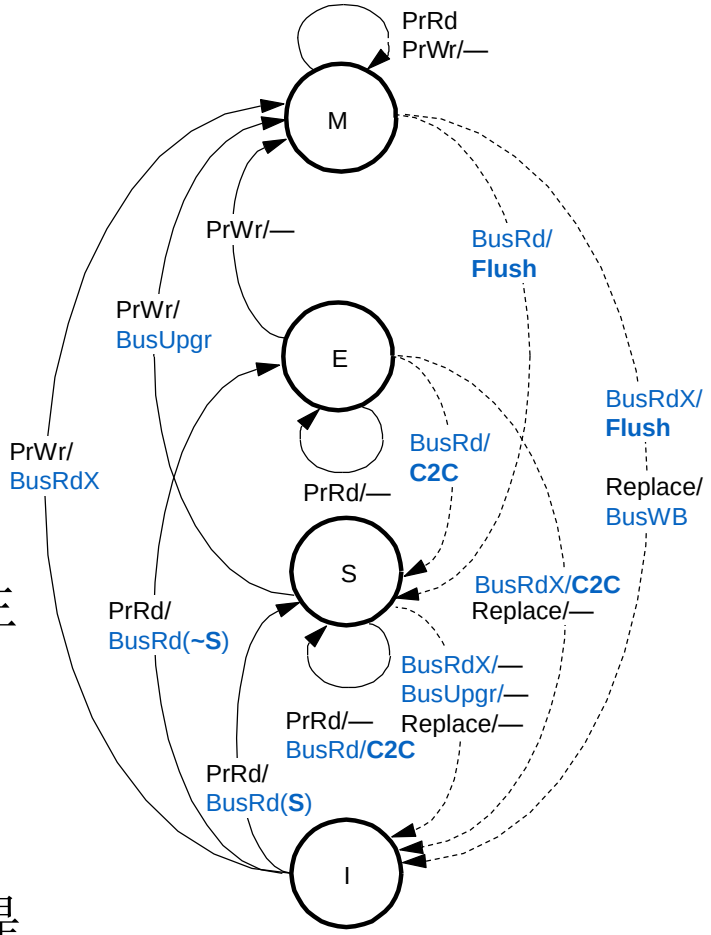
MESI State Transition Diagram

- Processor Read
  - 读失效时产生BusRd事务
  - BusRd(**S**) => shared line asserted
    - 在其他Cache中有有效的copy
    - Goto state *S*
  - BusRd(**~S**) => shared line not asserted
    - 在其他Cache中不存在该块
    - Goto state *E*
  - 读命中时不产生总线事务
- Processor Write
  - 该Cache块的状态转至 *M*
  - 在I或S态产生 BusRdX / BusUpgr
    - 作废其他Cache中的Copies
  - Cache块处于状态E and M时，不产生总线事务



MESI State Transition Diagram – cont'd

- Observing a BusRd
  - 该块的状态从E降至 *S*
    - 因为存在其他copy
  - 该块从*M降至S*
    - 将引起更新过的块刷新操作
    - 刷新内存和其他有需求的Cache
- Observing a BusRdX or BusUpgr
  - 将作废相应的 block
  - 对于处于modified状态的块，将产生flush事务
- Cache-to-Cache (**C2C**) Sharing
  - 原来的Illinois version支持这种共享
  - 由Cache 提供数据，而不是由内存提供数据



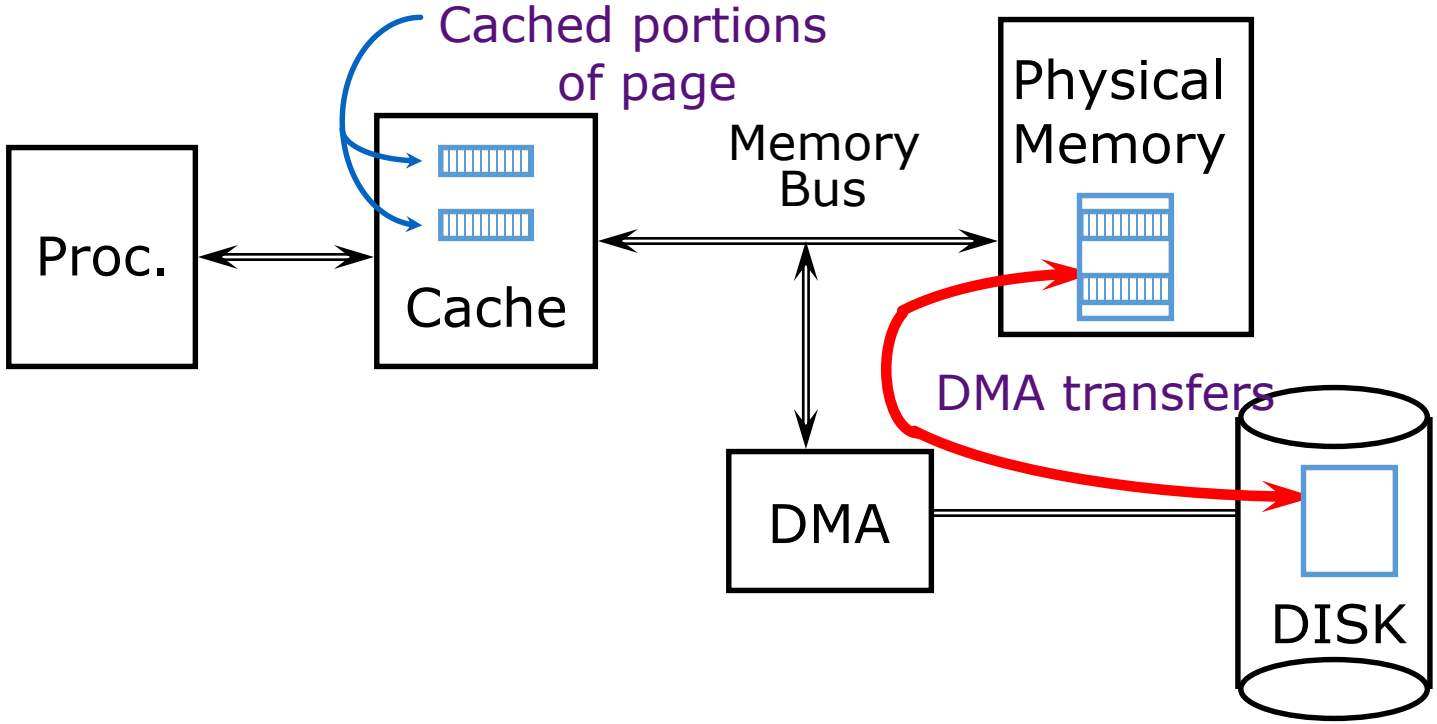
MESI Lower-level Design Choices

- 在E或S态时，由谁为BusRd/BusRdx事务提供数据
  - Original, Illinois MESI: cache, 因为假设Cache比memory更块
- 但 cache-to-cache 共享增加了实现的复杂性
  - 这种实现的代价高于从memory获取数据
  - 存储器如何知道它该提供数据 (must wait for caches)
  - 如果多个Cache共享数据，要有Selection
- 当块为Modified，总线上的刷新（Flushing）数据操作
  - 需要更新的块和存储器接收数据
  - 但存储器速度比Cache的速度慢，因此只有Cache接收数据，memory不接收数据
  - 这就要求第5个状态: **Owned** state ⇒ **MOESI** Protocol
  - Owned 态是共享的Modified态，此时存储器不是最新数据
    - 该块可以被多个Cache共享，但所有者（owner)只有一个

MOESI中的Owned 和Shared 状态

- Owned位。
  - ✓O位为1表示在当前Cache 块中包含的数据是当前处理器系统最新的数据拷贝，而且在其他CPU中一定具有该Cache行的副本，其他CPU的Cache行状态为S。
  - ✓如果存储器的数据在多个CPU的Cache中都具有副本时，有且仅有一个CPU的Cache行状态为O，其他CPU的Cache行状态只能为S。
  - ✓与MESI协议中的S状态不同，状态为O的Cache行中的数据与存储器中的数据并不一致。
- Shared位。
  - ✓当Cache块状态为S时，其包含的数据并不一定与存储器一致。
  - ✓如果在其他CPU的Cache中不存在状态为O的副本时，该Cache行中的数据与存储器一致；
  - ✓如果在其他CPU的Cache中存在状态为O的副本时，Cache块中的数据与存储器不一致。

Problems with Parallel I/O

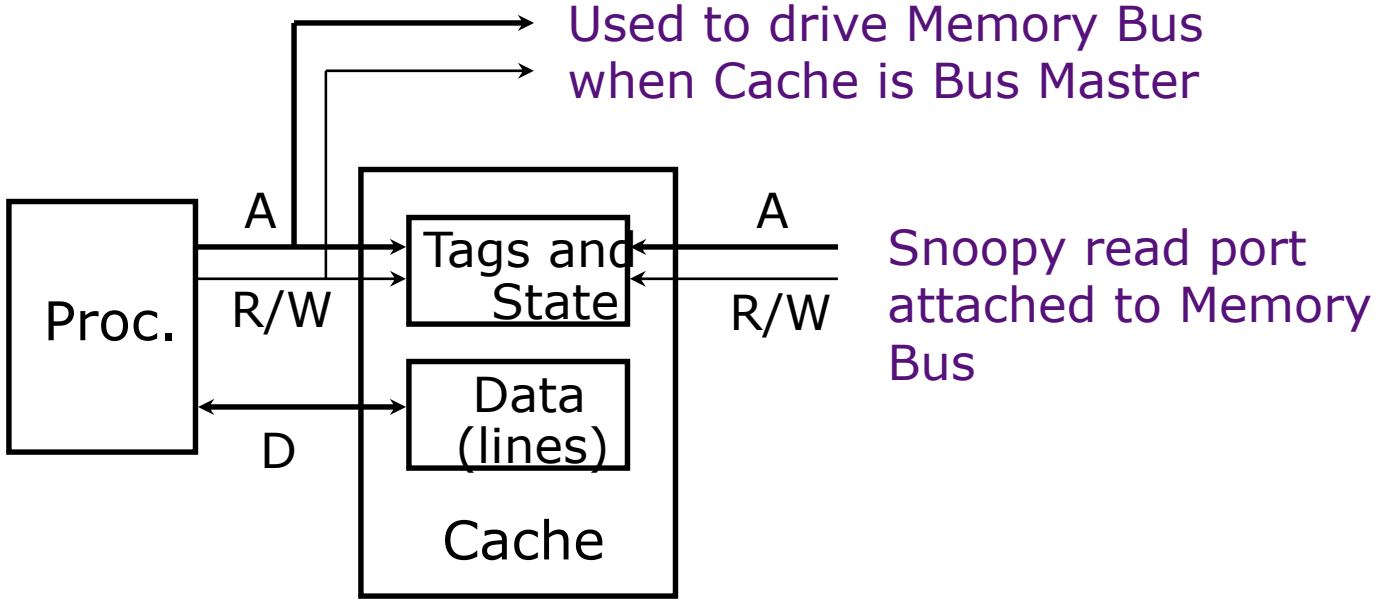


Memory → Disk: 如果cache的数据被修改过，而没有写回，存储器是陈旧数据

Disk → Memory: Cache中的数据是陈旧数据，它并不知道这次存储器写操作

Snoopy Cache, *Goodman 1983*

- Idea: 让Cache监测 DMA 传输, 然后 “do the right thing”
- 双端口的Snoopy cache tags



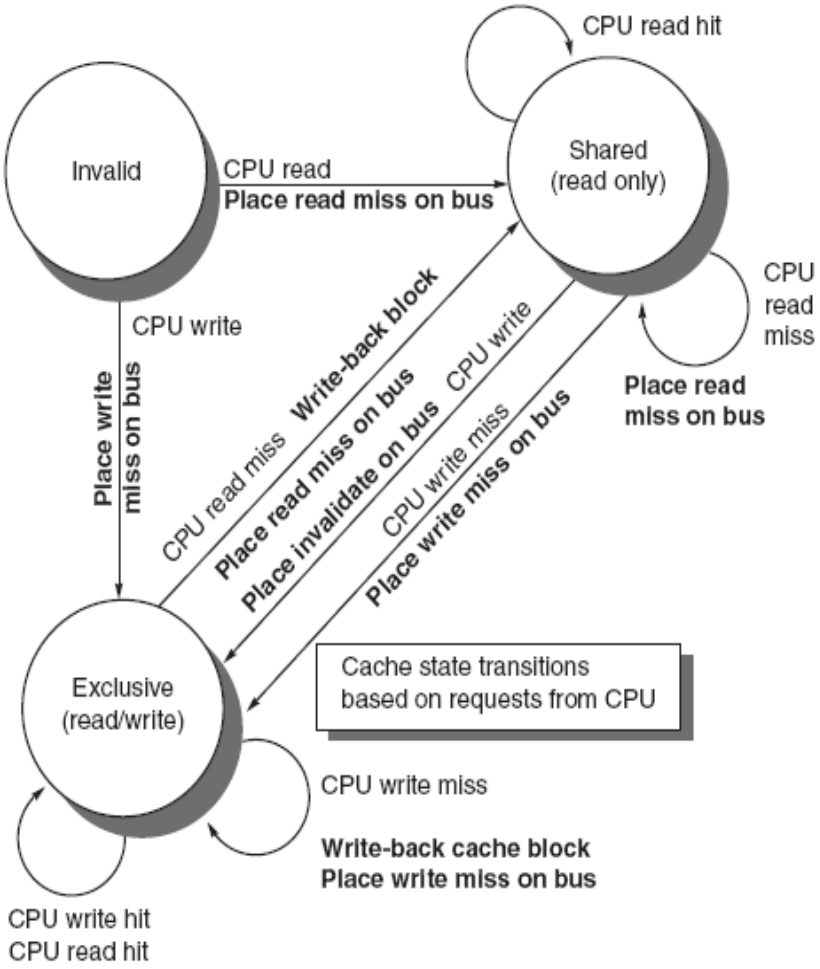
Snoopy Cache Actions for DMA

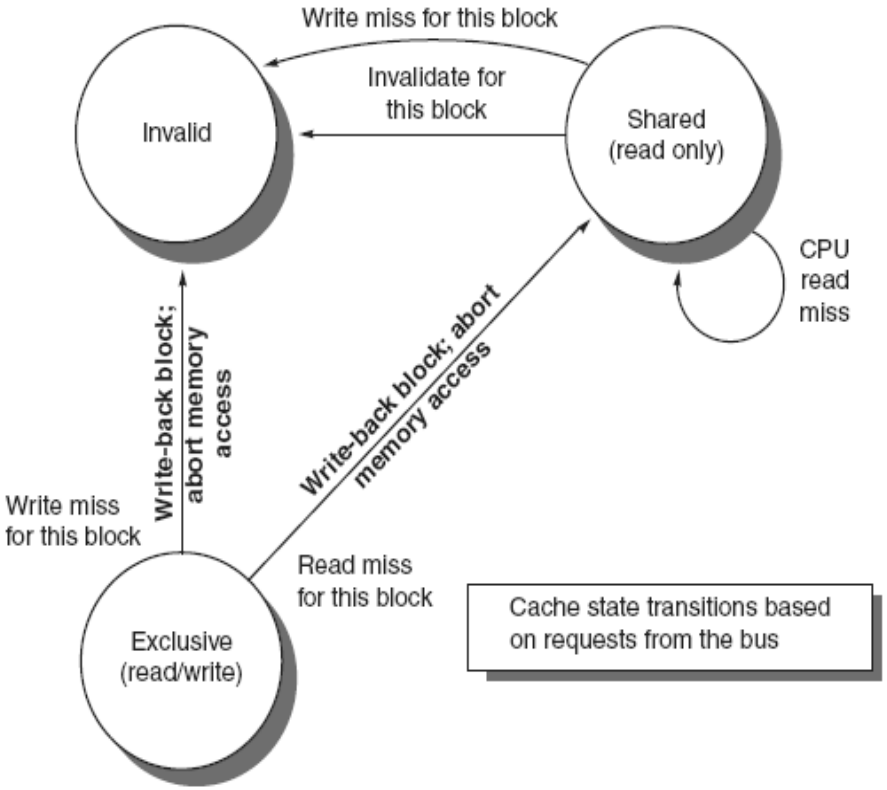
Observed Bus Cycle	Cache State	Cache Action
DMA Read Memory → Disk	Address not cached	No action
	Cached, unmodified	No action
	Cached, modified	Cache intervenes
DMA Write Disk → Memory	Address not cached	No action
	Cached, unmodified	Cache purges its copy
	Cached, modified	???

Snoopy Coherence Protocols

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Snoopy Coherence Protocols(MSI)





Performance of Symmetric Shared-Memory Multiprocessors

Cache performance 由两部分构成:

1. 单处理器 cache miss的通信量(Traffic)
2. 通信引起的通信量: 由于作废机制导致后面的访问失效

- *Coherence misses*
  - 有时也称为 *Communication* miss
  - 4th *C* of cache misses along with *C*ompulsory, *C*apacity, & *C*onflict.

Coherency Misses

- **由于对共享块的写操作引起**
  - 共享块在多个本地Cache有副本
  - 当某一处理器对共享块进行写操作时会 作废其他处理器的本地Cache的副本
  - 其他处理器对共享块进行读操作时 会有coherence miss
- **True sharing misses：Cache coherence 机制引起的数据通信**

  通常是不同的处理器写或读 同一个变量
  - 对S态块的写操作会作废其他cache中的共享块
  - 处理器试图读一个存在于其他处理器的cache中并且已经修改过的字（modified），这会导致失效，并将当前cache中的对应块写回
  - 即时块大小为1个字，失效仍然会发生
- **False sharing misses：由于某个字在某个失效块中**

  读写同一块中的不同变量
  - 失效并没有通过通信产生新的值，仅仅是产生了额外的失效
  - 块是共享的，但块中没有真正共享的字

    ⇒如果块的大小为1个字，那么就不会产生这种失效

Example of True & False Sharing Misses

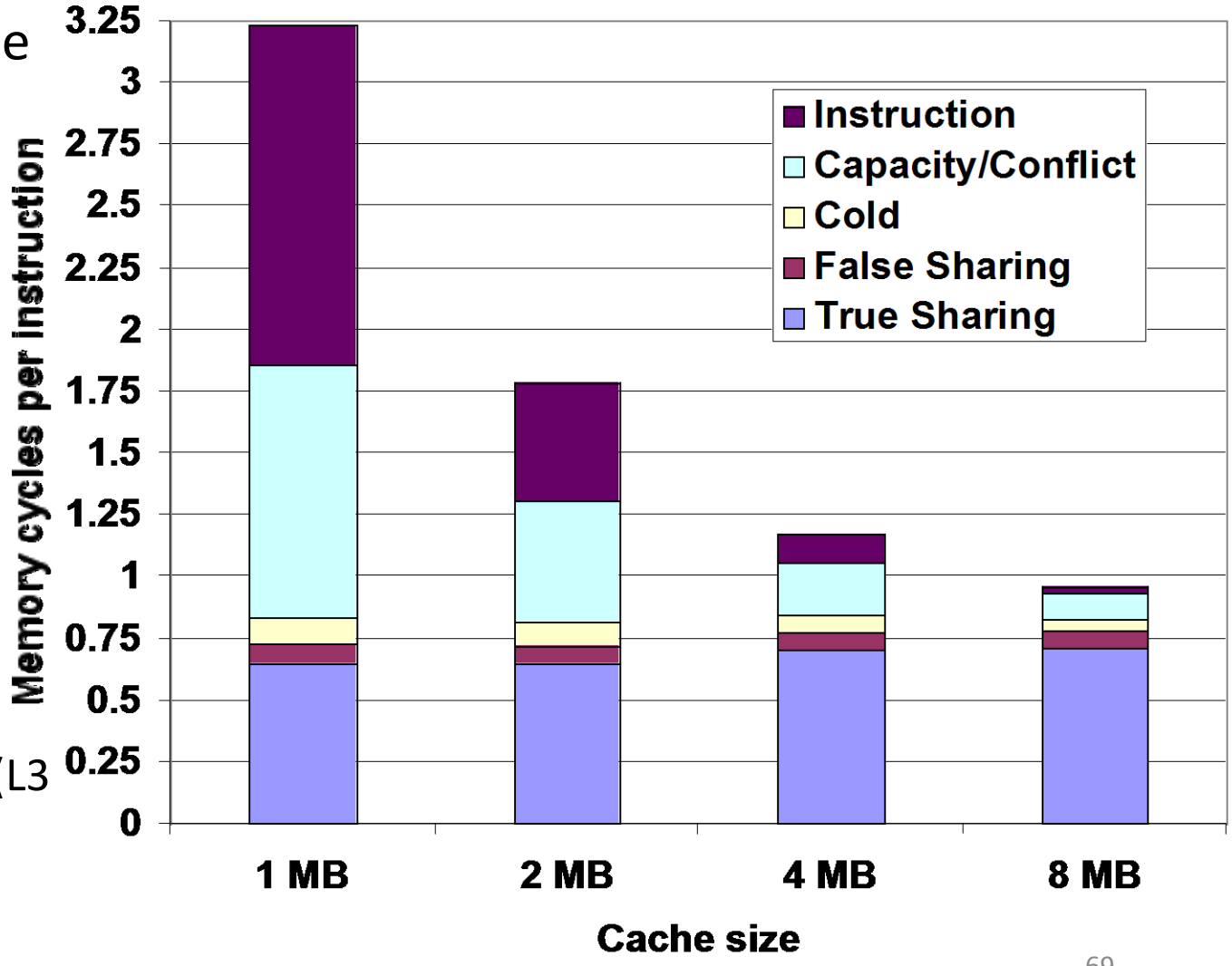
变量X和Y 属于同一cache块

初始状态为：P1和P2均读取了共享变量X, Block(X, Y) 在P1, P2中处于Shared 态

Request	P1 Cache State	P2 Cache State	Explanation
P1: Write X	Shared (X , Y)	Shared (X , Y)	**True Sharing Miss (P2 read X)**
	Modified (X , Y)	Invalid (X , Y)	P1 invalidates block (X , Y) in P2
P2: Read Y	Modified (X , Y)	Invalid (X , Y)	**False Sharing Miss (Y not modified)**
	Shared (X , Y)	Shared (X , Y)	Write-Back & Copy block from P1 to P2
P1: Write X	Shared (X , Y)	Shared (X , Y)	**False Sharing Miss (P2 did not read X)**
	Modified (X , Y)	Invalid (X , Y)	P1 invalidates block (X , Y) in P2
P2: Write Y	Modified (X , Y)	Invalid (X , Y)	**False Sharing Miss (P1 did not read Y)**
	Invalid (X , Y)	Modified (X , Y)	Write-Back & Copy block from P1 to P2
P1: Read Y	Invalid (X , Y)	Modified (X , Y)	**True Sharing Miss (P2 modified Y)**
	Shared (X , Y)	Shared (X , Y)	Write-Back & Copy block from P2 to P1

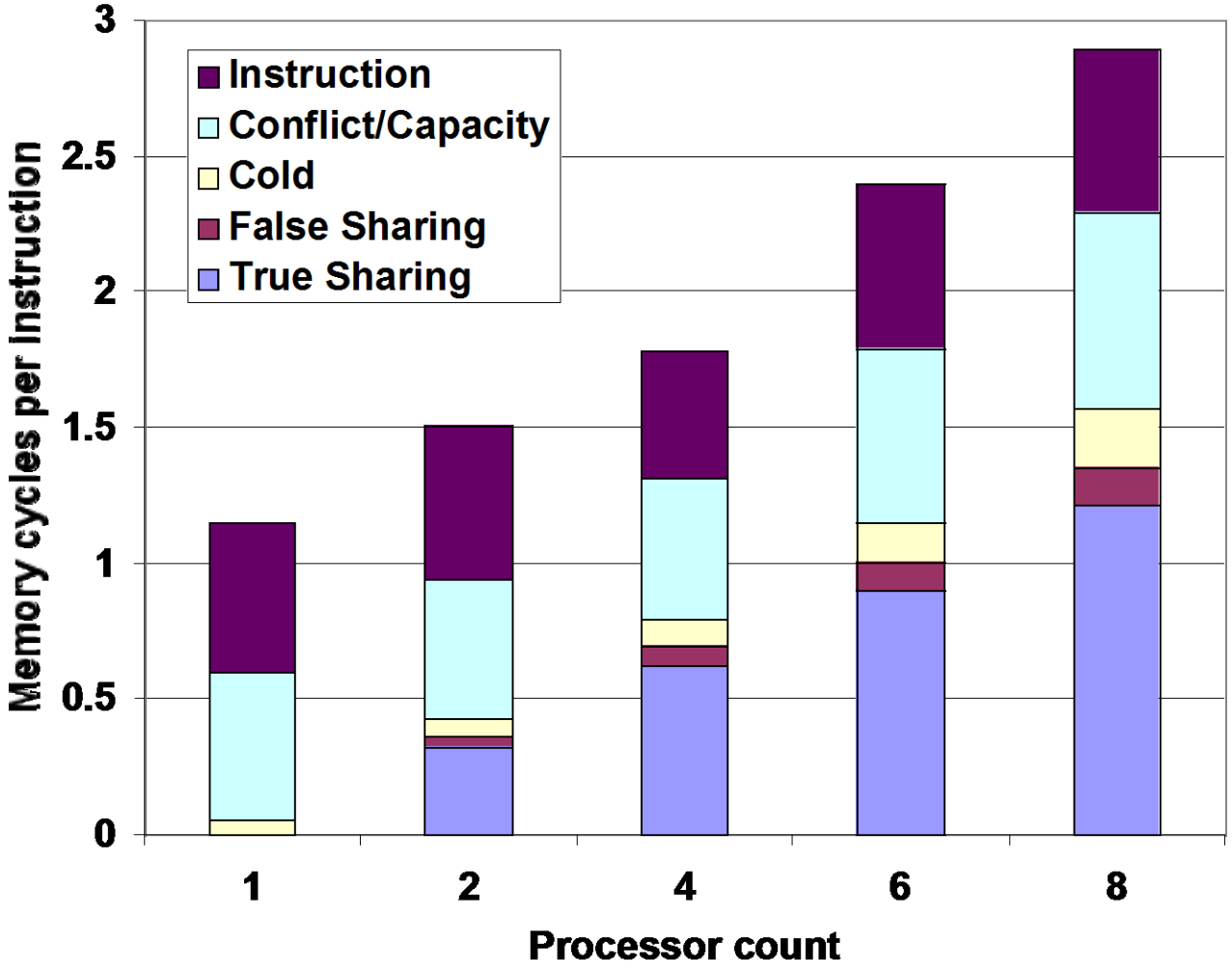
MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)
- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)



MP Performance 2MB Cache
Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs



7.3 分布式共享存储器体系结构

存储器分布于各结点中，所有的结点通过网络互连。访问可以是本地的，也可是远程的。

不支持Cache一致性：规定共享数据不进入Cache，仅私有数据才能保存在Cache中。

优点：所需的硬件支持很少(因为远程访问存取量仅是一个字(或双字)而不是一个Cache块)

不支持Cache一致性的缺点：

(1) 实现透明的软件Cache一致性的编译机制能力有限。

(2) 没有Cache一致性，机器就不能利用取出同一块中的多个字的开销接近于取一个字的开销这个优点。这是因为共享数据是以Cache块为单位管理的。当每次访问要从远程存储器取一个字时，不能有效利用共享数据的空间局部性。

(3) 诸如预取等延迟隐藏技术对于多个字的存取更为有效，比如针对一个Cache块的预取。

Limitations of Snooping Protocols

- 总线的可扩放性收到一定限制
  - 总线上能够连接的处理器数目有限
  - 共享总线存在竞争使用问题
  - 在由大量处理器构成的多处理器系统中，监听带宽是瓶颈
- 片上互连网络→并行通信
  - 多个处理器可并行访问共享的Cache banks
  - 允许片上多处理器包含有更多的处理器
- 在非总线或环的网络上监听是比较困难的
  - 必须将一致性相关信息广播到所有的处理器，这是比较低效的
- 如何不采用广播方式而保持 cache coherence
  - 使用目录（directory)来记录每个 Cached 块的状态
  - 目录项说明了哪个私有Cache包含了该块的副本

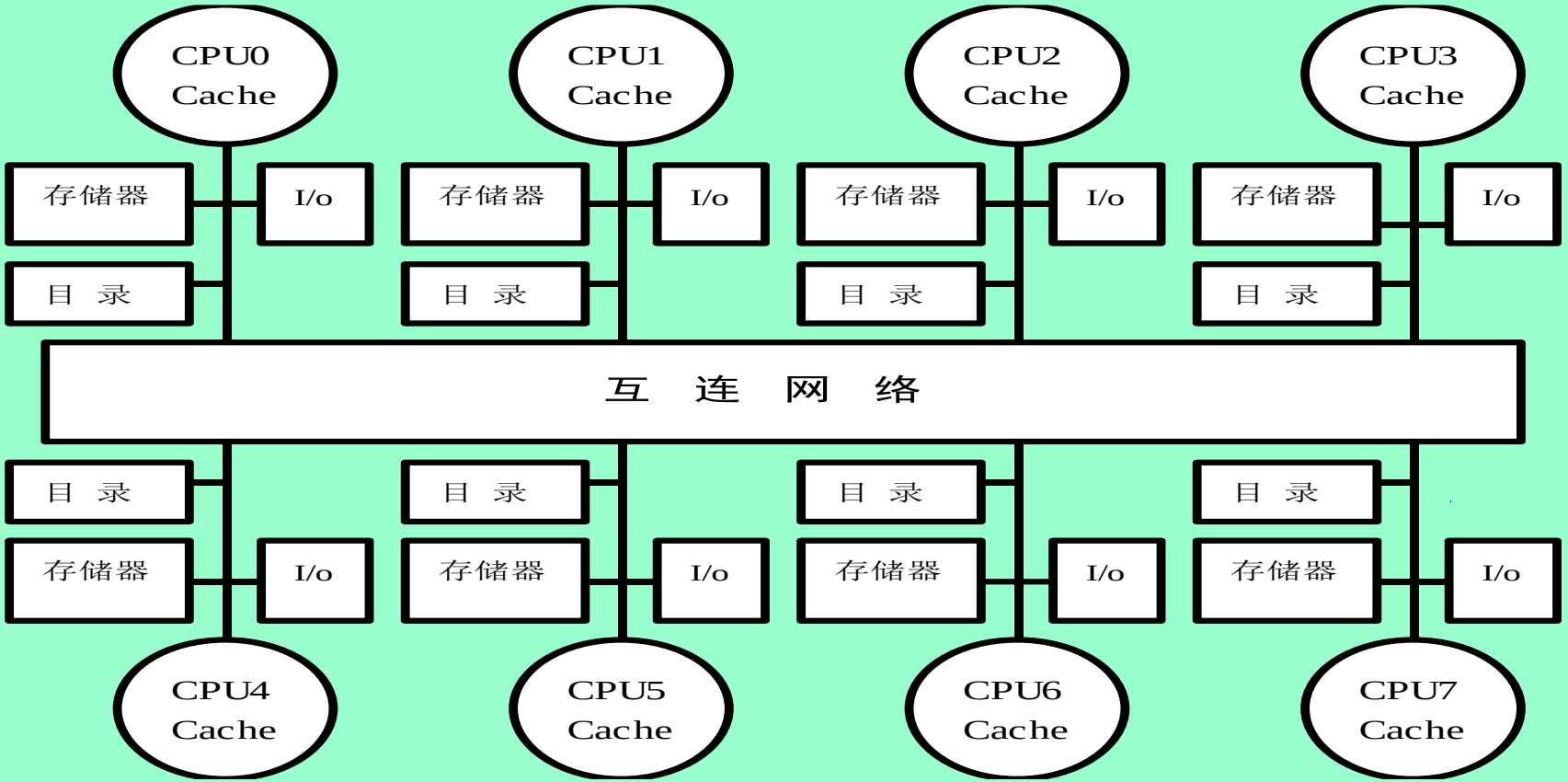
解决Cache一致性问题的关键

寻找替代监听协议的一致性协议。

- 目录协议

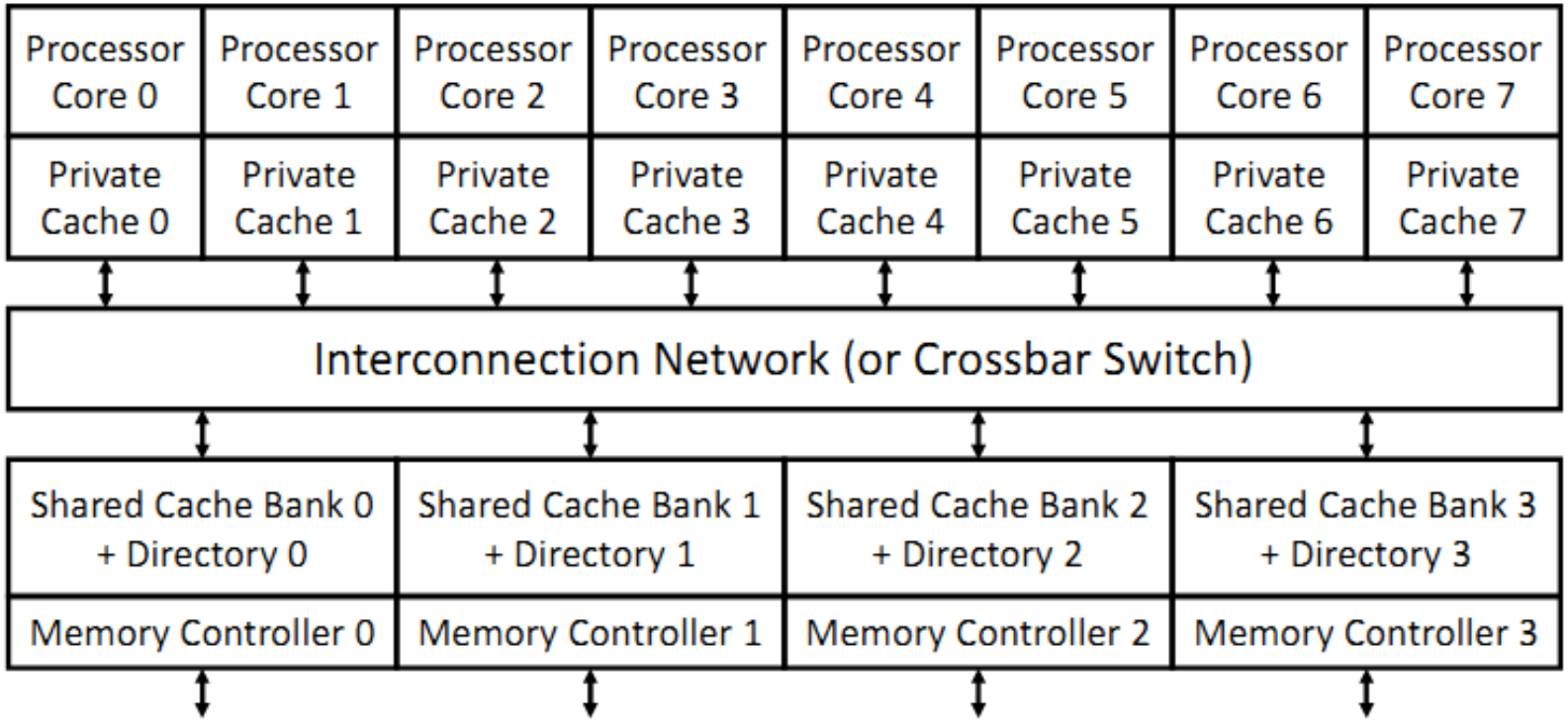
目录：用于记录共享块相关信息的数据结构，它记录着可以进入Cache的每个数据块的访问状态、该块在各个处理器的共享状态以及是否修改过等信息。

- 对每个结点增加目录表后的分布式存储器的系统结构


CPU0
Cache
CPU1
Cache
CPU2
Cache
CPU3
Cache
存储器
I/o
目 录
互 连 网 络
CPU4
Cache
CPU5
Cache
CPU6
Cache
CPU7
Cache

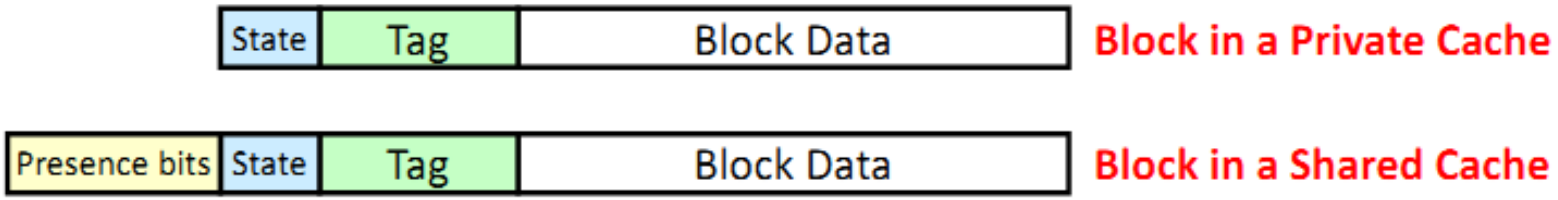
Directory in a Chip Multiprocessor

- 目录在所有处理器共享的最外层Cache中
  - 目录记录了每个私有Cache中块的相关信息
- 最外层Cache分成若干个banks，以便并行访问
  - Cache的banks数可以与cores的数量相同，也可以不同



Directory in the Shared Cache

- Shared Cache 包含所有的私有Cache
  - 共享Cache是私有cache块的超集
  - Example: Intel Core i7
- 目录在共享cache中
  - 共享cache中的每个块增加若干presence bits
  - 如果有k个processors那么共享cache中每个块含有presence bits(k位) + state位
  - Presence bits 指示了包含该块copy的cores
  - 每个块都有其在私有cache和共享cache中的状态信息
  - State = M (Modified), S (Shared), or I (Invalid) in private cache



一些术语

- 本地或私有Cache （Local (or Private) Cache）
  - 处理器请求的源
- 目录（Home Directory）
  - 存放Cache块相关信息
  - 目录使用presence bits 和 state 追踪cache块
- 远程Cache（Remote Cache）
  - 该Cache中包含一个Cache块的副本，处于modified 或shared 态
- Cache一致性 即要保证Single-Writer, Multiple-Readers
  - 如果一个块在本地Cache中处于Modified态，那么只有一个有效的副本存在
    - 共享的Cache和存储器还没有更新
- 无总线，不想用广播方式让所有处理器核
  - 所有消息都有显式的回复

States for Local and Shared Cache

对于本地（私有）cache 块，存在3种状态:

1. Modified: 仅当前Cache具有该块修改过的副本
2. Shared: 该块可能在多个Cache中有副本
3. Invalid: 该块无效

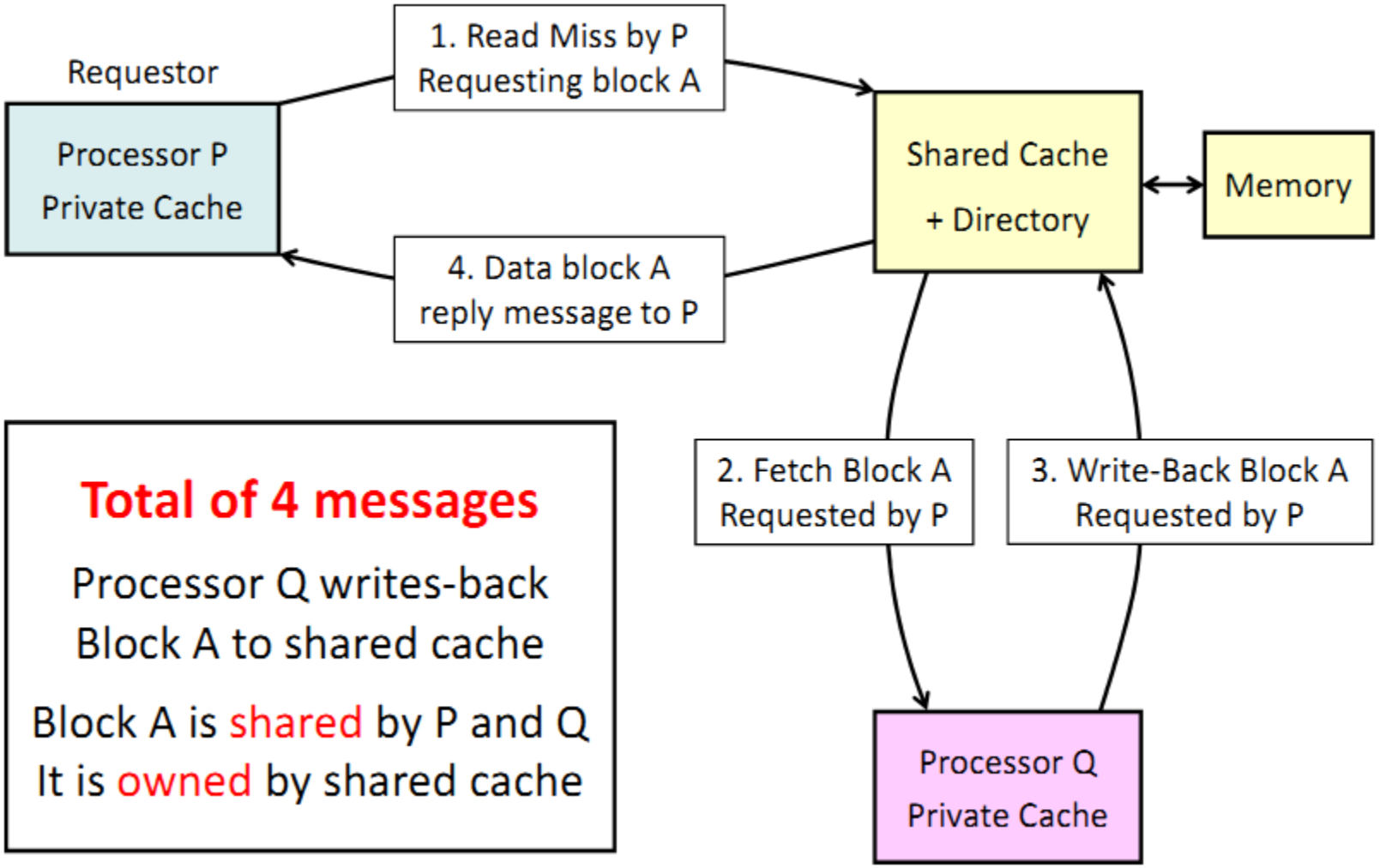
对于共享Cache中的块，存在4种状态:

1. Modified: 只有一个本地Cache是这个块的拥有者
   - 只有一个本地Cache具有该块修改后的副本
2. Owned: 共享Cache是modified块的拥有者
   - Modified block被写回到共享Cache，但不是内存
   - 处于owned态的块可以被多个本地Cache共享
3. Shared: 该块可能被复制到多个cache中
4. Uncached: 该块不在任何本地或共享Cache中

Read Miss by Processor P

- Processor P 发送 Read Miss 消息给 Home directory
- Home Directory: block 是 Modified态
  - Directory 发送 Fetch message 到拥有modified块的 remote cache
  - Remote cache发送 Write-Back message 到 directory (shared cache)
  - Remote cache 将该块状态修改为shared
  - Directory 将其所对应的共享块状态修改为 owned
  - Directory 发送数据给P, 并将处理器P的presence bit置位
  - 处理器P的Local cache 将所接收到的块状态置为 shared
- Home Directory: block 是Shared or Owned态
  - Directory发送数据给处理器P，并将对应 处理器P的presence bit置位
  - 处理器P的Local cache 将所接收到的块状态置为 shared
- Home Directory: Uncached -> 从存储器中获取块

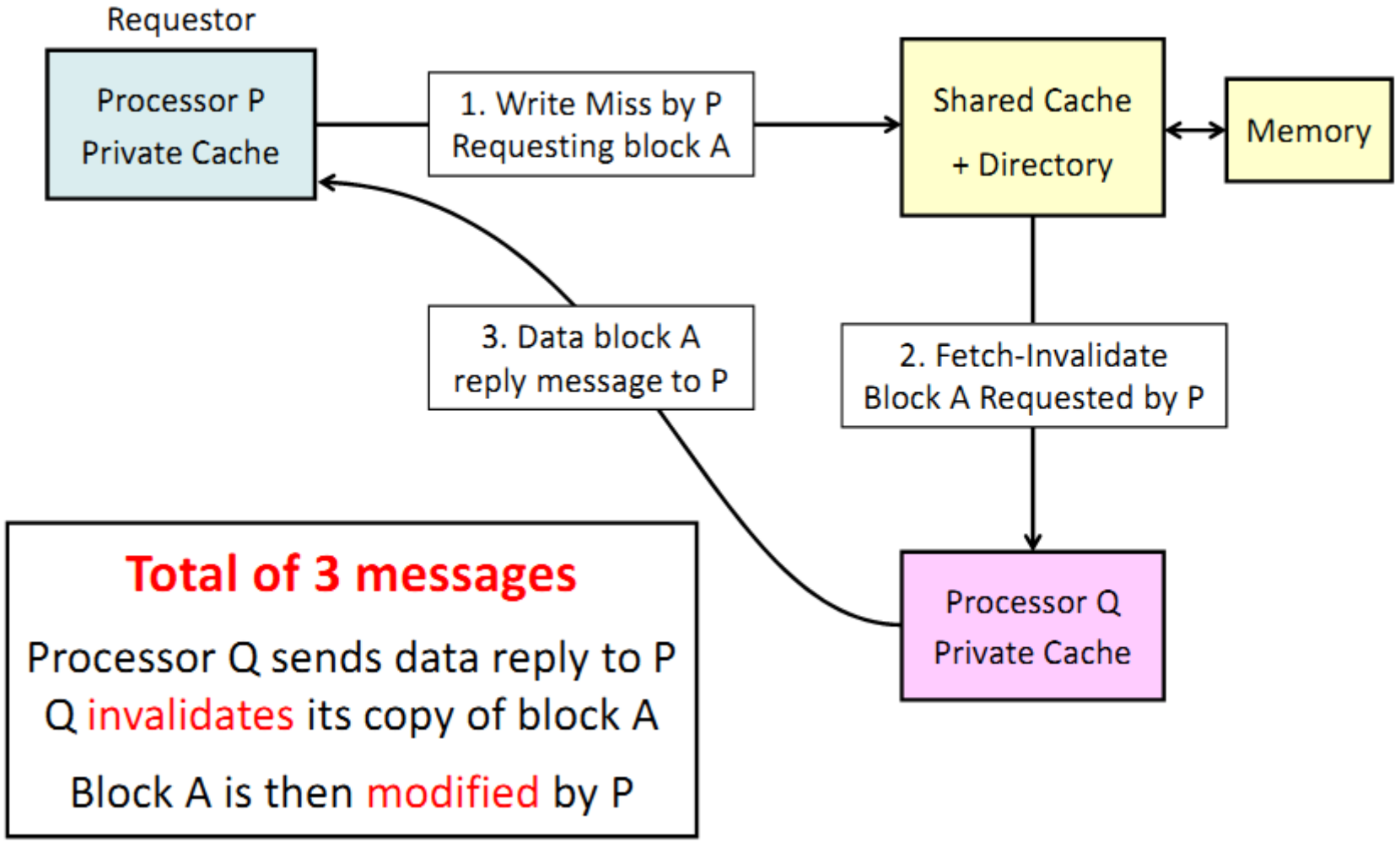
Read Miss to a Block in Modified State



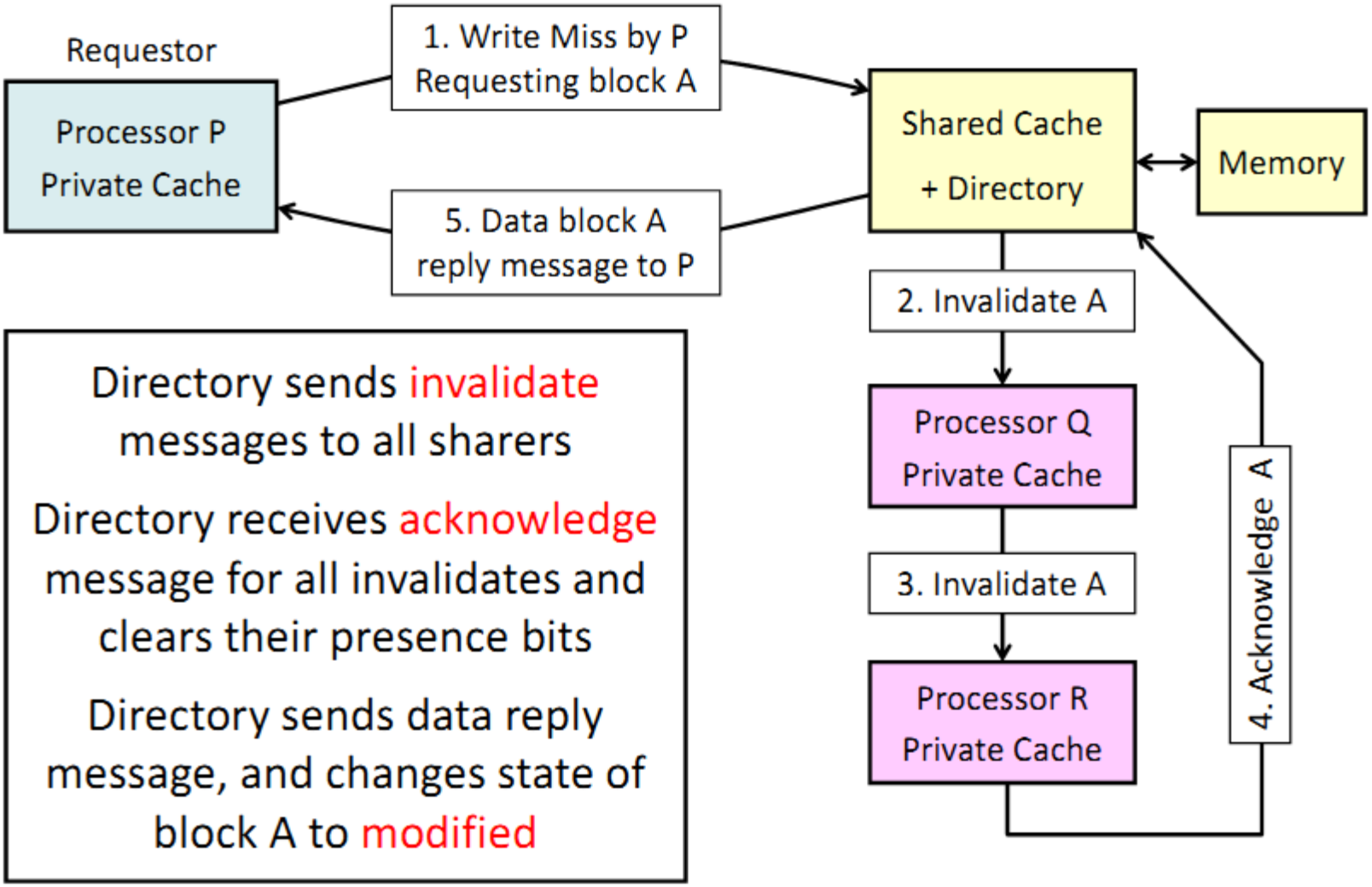
Write Miss Message by P to Directory

- Home Directory: block 是Modified态
  - Directory 发送 Fetch-Invalidate message 给处理器Q的远程Cache
  - 处理器Q的Remote cache 直接发送数据应答消息给P
  - Remote cache将对应块的状态修改为invalid
  - P的Local cache 将接收到的块的状态信息修改为modified
  - Directory 将对应于Q的 presence bit复位，并将对应于P的presence bit 置位
- Home Directory: block 是 Shared or Owned态
  - Directory 根据presence bit位给所有的共享者发送invalidate messages
  - Directory接收 acknowledge消息并将对应的presence bits复位
  - Directory 发送数据回复信息给P, 并将P对应的 presence bit置位
  - P的Local cache 和directory 将该块的状态修改为 modified
- Home Directory: Uncached -> 从存储器获取数据

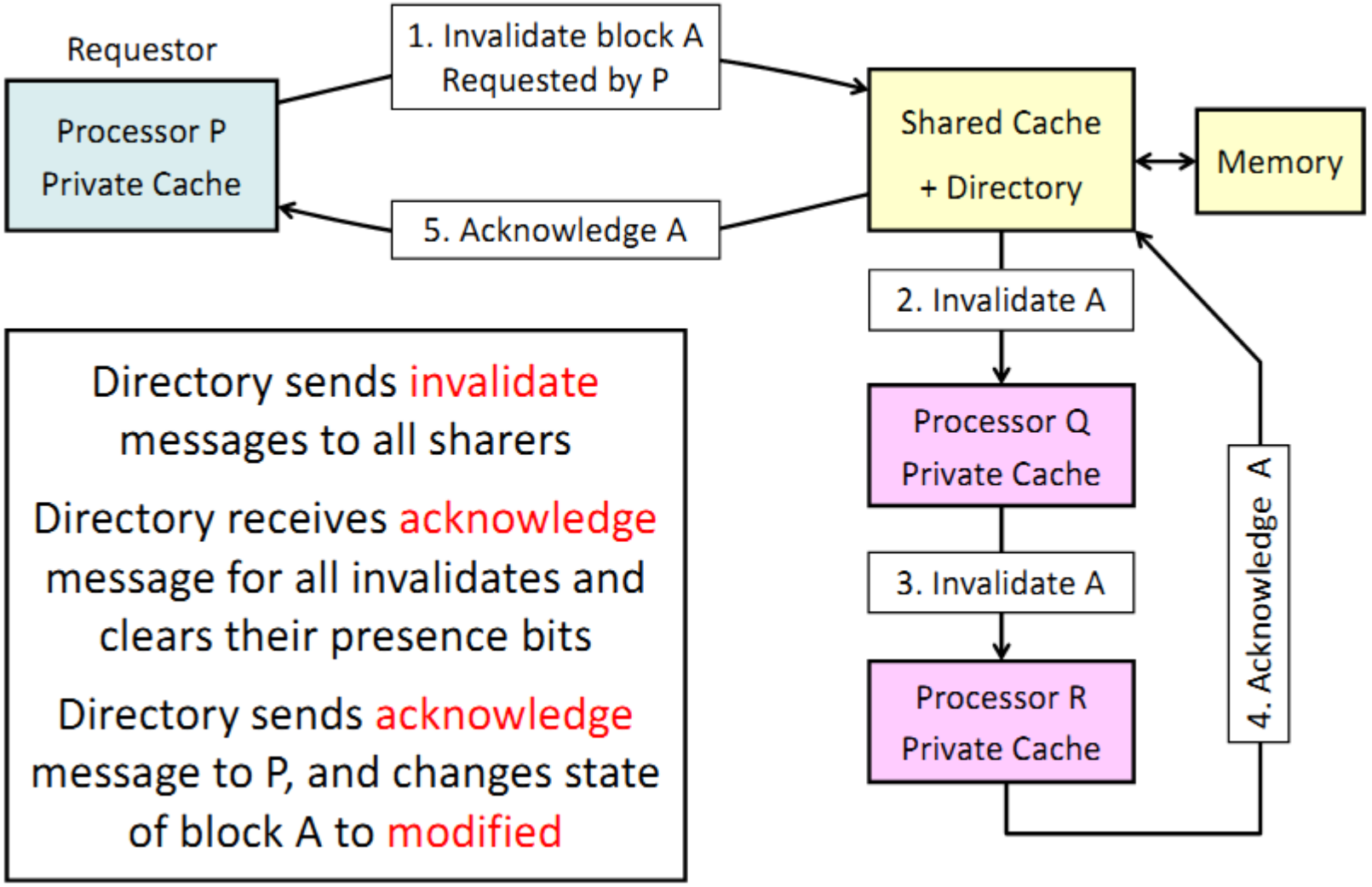
Write Miss to a Block in Modified State



Write Miss to a Block with Sharers



Invalidating a Block with Sharers

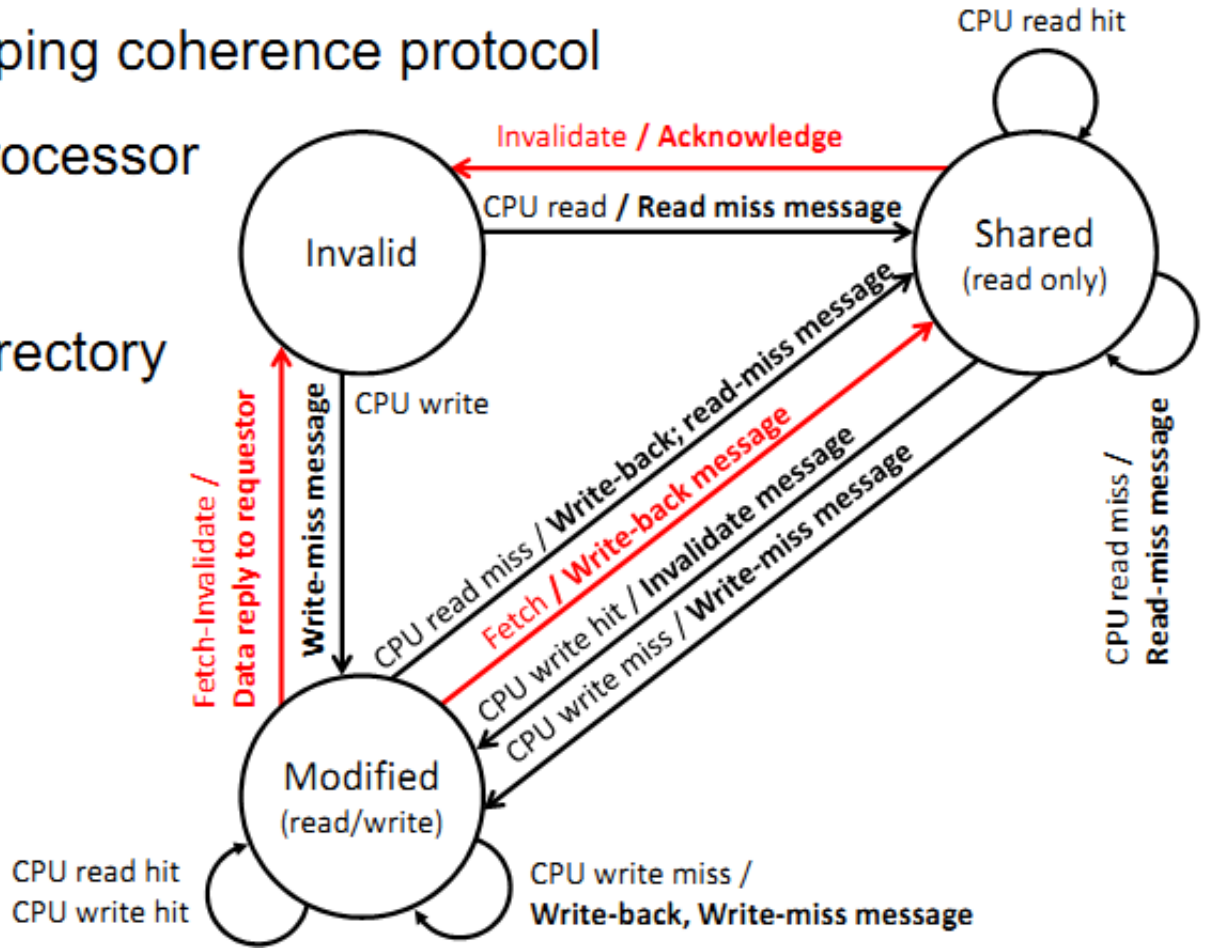


Directory Protocol Messages

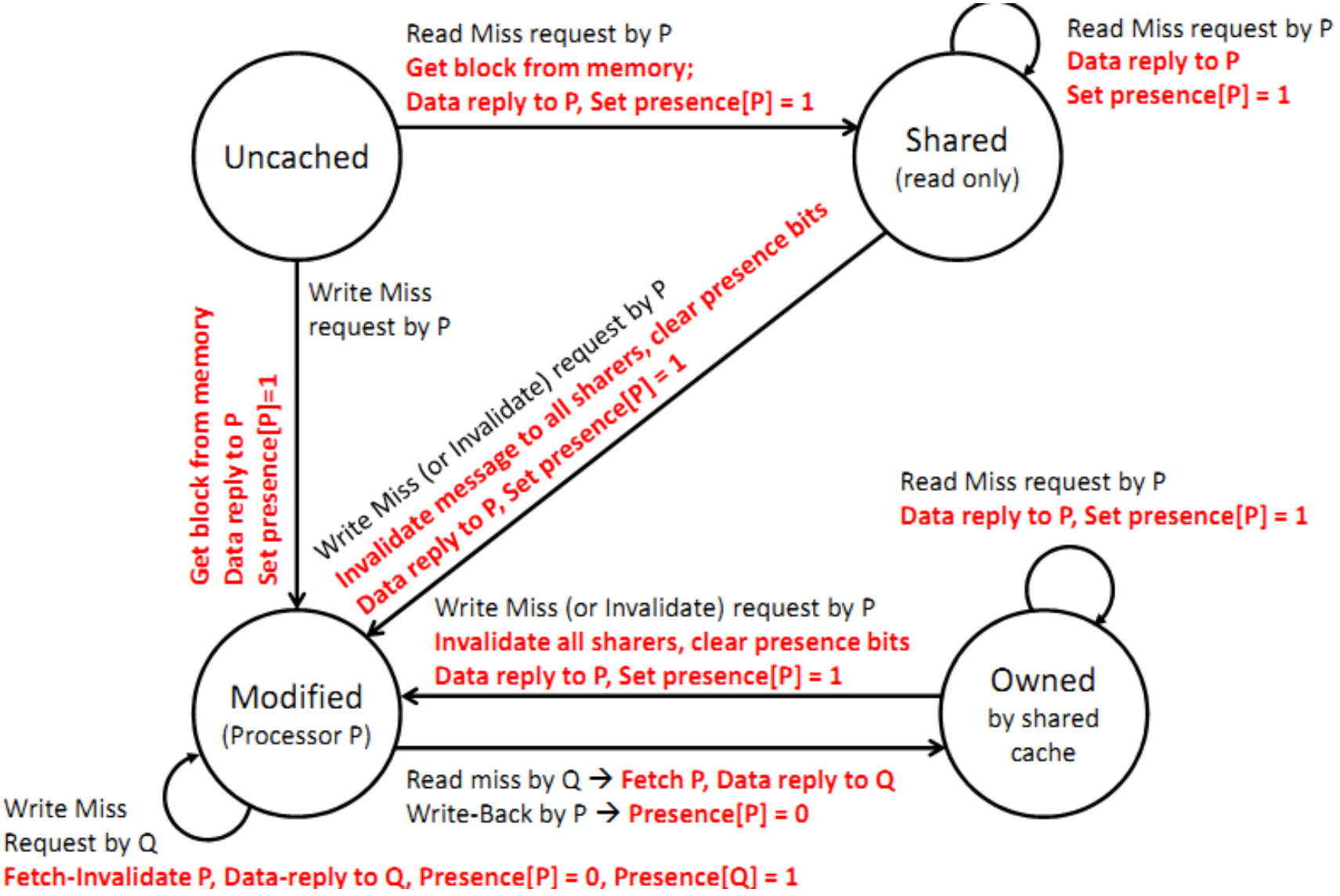
Message Type	Source	Destination	Message Function
Read Miss	Local Cache	Home Directory	Processor P has a read miss at address A Request data and make P a read sharer
Write Miss	Local Cache	Home Directory	Processor P has a write miss at address A Request data and make P the exclusive owner
Invalidate	Local Cache	Home Directory	Processor P wants to invalidate all copies of the same block at address A in all remote caches
Invalidate	Home Directory	Remote Caches	Directory sends invalidate message to all remote caches to invalidate shared block at address A
Acknowledge	Remote Cache	Home Directory	Remote cache sends an acknowledgement message back to home directory after invalidating last shared block A
Acknowledge	Home Directory	Local Cache	Directory sends acknowledgment message back to local cache of P after invalidating all shared copies of block A
Fetch	Home Directory	Remote Cache	Directory sends a fetch message to a remote cache to fetch block A and to change its state to shared
Fetch & Invalidate	Home Directory	Remote Cache	Directory sends message to a remote cache to fetch block A and to change its state to invalid
Data Block Reply	Directory or Cache	Local Cache	Directory or remote cache sends data block reply message to local cache of processor P that requested data block A
Data Block Write Back	Remote Cache	Home Directory	Remote Cache sends a write-back message to home directory containing data block A

MSI State Diagram for a Local Cache

- Three states for a cache block in a local (private) cache
- Similar to snooping coherence protocol
- Requests by processor
  - **Black arrows**
- Requests by directory
  - **Red arrows**



MOSI State Diagram for Directory



7.4 Models of Memory Consistency

- 什么是存储同一性？
- 隐式存储同一性模型（顺序存储同一性）
- 放松的存储同一性

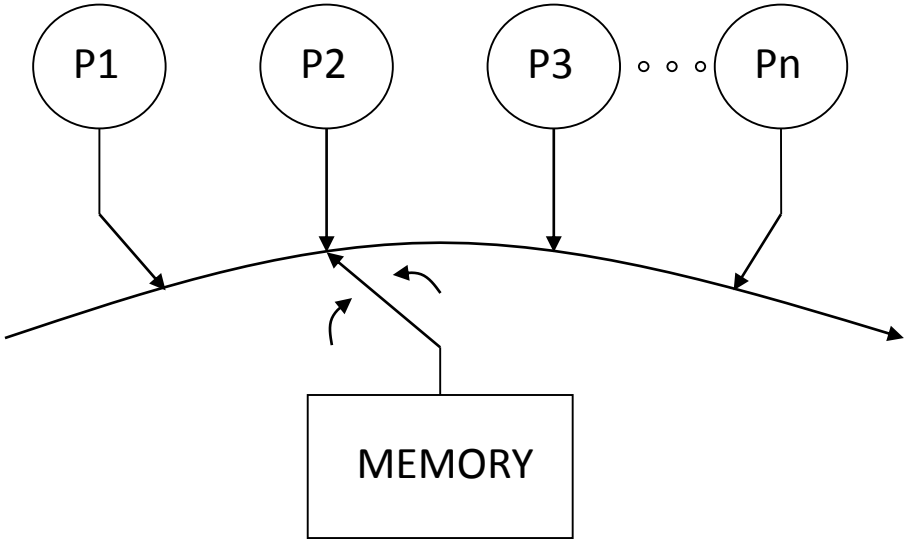
存储同一性的定义

- 定义: 共享地址空间的存储同一性模型是在多个处理器对不同存储单元并发读写操作时，每个进程看到的这些操作被完成的序的一种约定。
  - ✓ 存储一致性保证的是当对共享存储空间中的某一单元修改后，对所有读取者是可见的。即每个单元都能“返回最后一次写操作的值”
  - ✓ 没有明确所写入的数据何时成为可见的
  - ✓ 一致性协议没有涉及处理器P1和P2对不同地址单元的访问顺序
  - ✓ 一致性协议没有涉及到P2对不同存储单元的读操作相对于P1所见到的顺序

```
P1                  P2        (A, flag are zero initial)
A=1                 while(flag == 0);
flag=1              print A;
```

Implicit Memory Model

- 顺序同一性(Sequential Consistency) [Lamport]：该模型要求所有处理器的读、写和交换(swap)操作串行执行所形成的全局存储器次序，符合各处理器的原有程序次序。即： 不论指令流如何交换执行，全局次序必须保持所有的程序次序
  - ✓所有读写操作执行以某种顺序执行
  - ✓每一进程的操作以程序序执行



- No caches, no write buffers

Understanding Program Order – Example 1

• Initially X = 2	
• P1	P2
• …..	…..
• r0=Read(X)	r1=Read(x)
• r0=r0+1	r1=r1+1
• Write(r0,X)	Write(r1,X)
• …..	……

- Possible execution sequences:

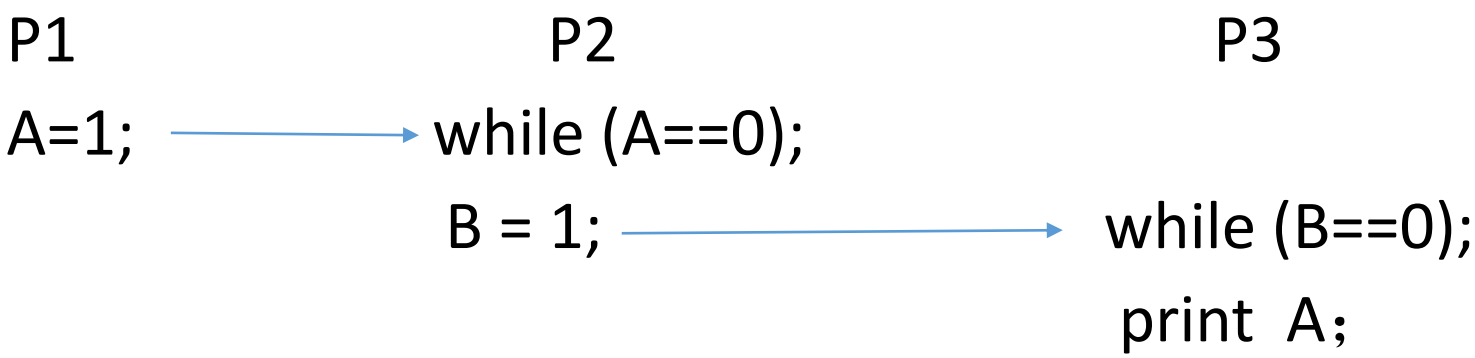
• P1:r0=Read(X)	P2:r1=Read(X)
• P2:r1=Read(X)	P2:r1=r1+1
• P1:r0=r0+1	P2:Write(r1,X)
• P1:Write(r0,X)	P1:r0=Read(X)
• P2:r1=r1+1	P1:r0=r0+1
• P2:Write(r1,X)	P1:Write(r0,X)
• x=3	x=4

Atomic Operations

- 顺序一致性并不保证操作的原子性
- 原子性：存储器操作使用原子性操作，该操作或操作序列一次全部完成。例如exchange（交换操作）。
  - exchange(r,M): 互换寄存器r与存储单元M的内容

```
r0 = 1;
do exchange(r0,S) while (r0 != 0); //S is memory location
 //enter critical section
 .....
 //exit critical section
 S = 0;
```

Understanding Program order-Example0



假设A,B的初始值为0；

从程序员角度看；P3应该输出 A=1；

如果P2被允许越过对变量A的读操作，在P3看见A的新值前对B进行写操作，那么P3就可能读出B的新值和A的旧值（例如从cache)，这种情况就不满足顺序同一性要求。

Understanding Program Order – Example 1

Initially Flag1 = Flag2 = 0

P1

Flag1 = 1

if (Flag2 == 0)

critical section

P2

Flag2 = 1

if (Flag1 == 0)

critical section

Execution:

P1	P2
(Operation, Location, Value)	*(Operation, Location, Value)*
Write, Flag1, 1	Write, Flag2, 1
Read, Flag2, 0	Read, Flag1, ___

Understanding Program Order – Example 1

- Initially Flag1 = Flag2 = 0
- P1 P2
- Flag1 = 1 Flag2 = 1
- if (Flag2 == 0) if (Flag1 == 0)
- *critical section* *critical section*

- Execution:
- P1 P2
- *(Operation, Location, Value)* *(Operation, Location, Value)*
- Write, Flag1, 1 Write, Flag2, 1
- Read, Flag2, 0 Read, Flag1, ____

Understanding Program Order – Example 1

Initially Flag1 = Flag2 = 0

P1

Flag1 = 1

if (Flag2 == 0)

critical section

P2

Flag2 = 1

if (Flag1 == 0)

critical section

Execution:

P1

(Operation, Location, Value)

Write, Flag1, 1

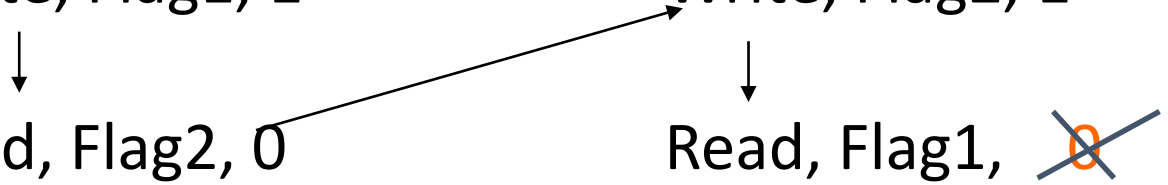
Read, Flag2, 0

P2

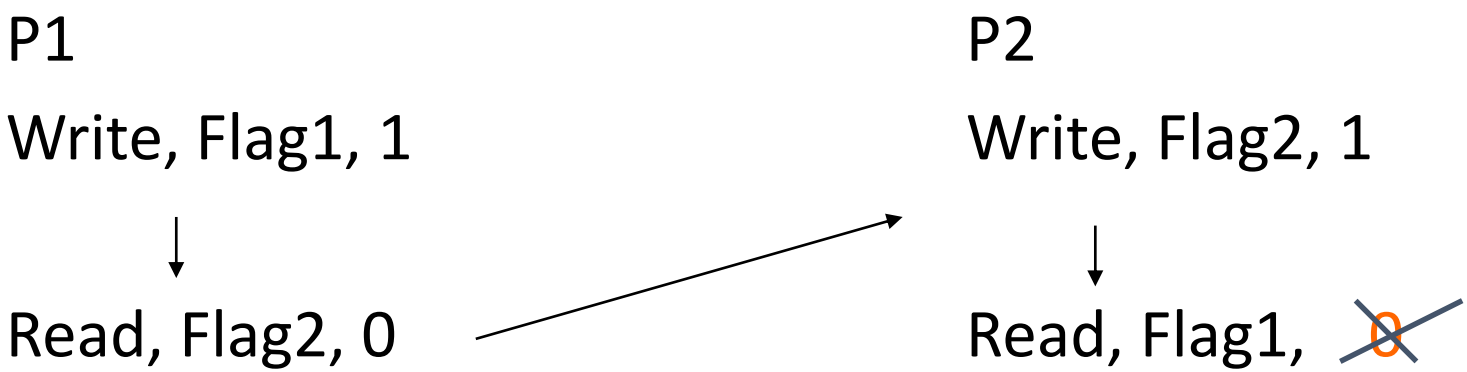
(Operation, Location, Value)

Write, Flag2, 1

Read, Flag1, 0



Understanding Program Order – Example 1



- ✓当我们不保证写操作的原子性时（有write buffer），就可能会发生P2读Flag1为0。
- ✓如果P2和P1交织执行，硬件或编译重排序的读写序，也会有这种情况。
- ✓当我们不保证P1和P2的读写序列的原子性时，就有可能发生两进程都无法进入临界区的问题。

Understanding Program Order - Example 2

Initially A = Flag = 0

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {;}
... = A;

P1
Write, A, 23
Write, Flag, 1

P2
Read, Flag, 0

Read, Flag, 1
Read, A, ____

Understanding Program Order - Example 2

Initially A = Flag = 0

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {;}
... = A;

P1
Write, A, 23
Write, Flag, 1

P2
Read, Flag, 0

Read, Flag, 1
Read, A, ~~0~~

Understanding Program Order - Example 2

Initially A = Flag = 0

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {;}
... = A;

P1
Write, A, 23
Write, Flag, 1

P2
Read, Flag, 0

Read, Flag, 1
Read, A, ~~0~~

✓两进程交叉执行或硬件/编译器重定序了读写顺序

顺序同一性的充分条件

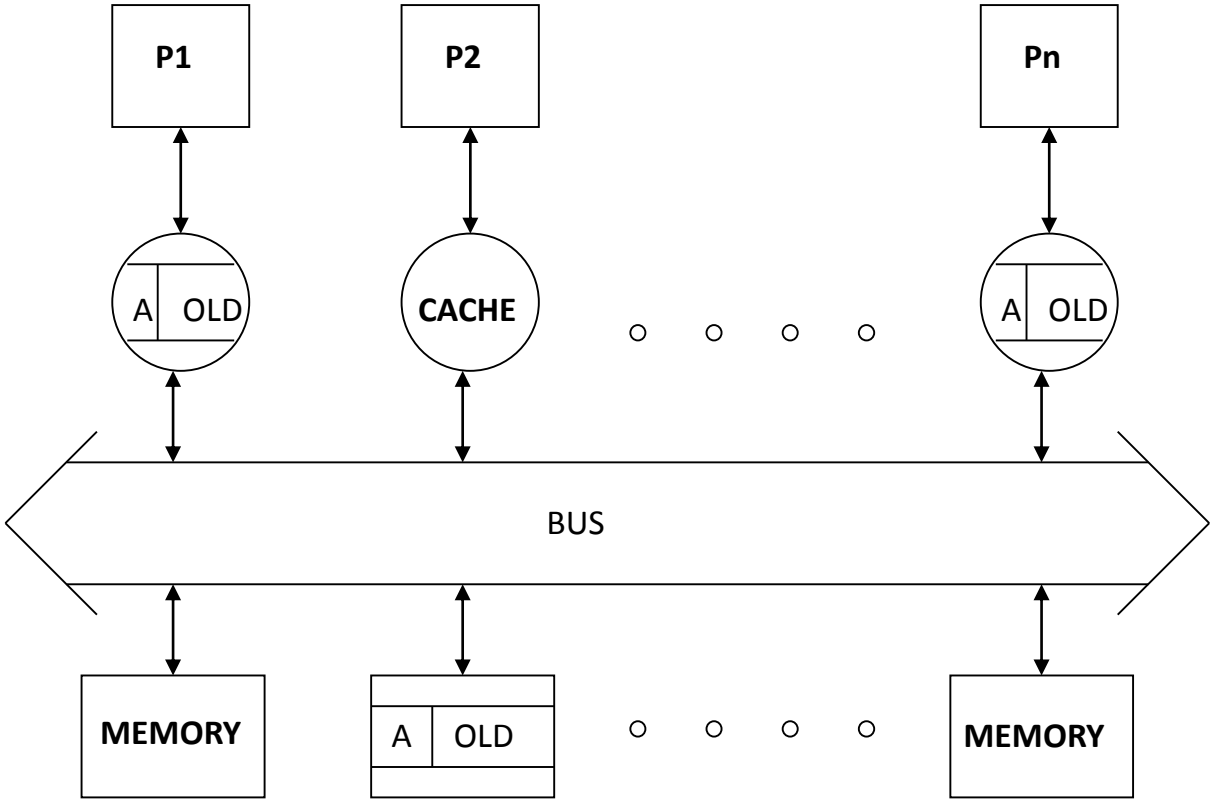
- 多个进程可以交织执行，但顺序同一性模型没有定义具体的交织方式，满足每个进程操作偏序的总体程序执行序可能会很多。因此有下列定义：

- 顺序同一性的执行：如果程序的一次执行产生的结果与前面定义的任意一种可能的总体顺序产生的结果一致，那么程序的这次执行就称为是顺序同一的。
- 顺序同一性的系统：如果在一个系统上的任何可能的执行都是顺序同一的，那么这个系统就是顺序同一的

顺序同一性的充分条件

- 每个进程按照程序执行序发出存储操作
- 发出写操作后，进程要等待写的完成，才能发出它的下一个操作
- 发出读操作后，进程不仅要等待读的完成，还要等待产生所读数据的那个写操作完成，才能发出它的下个操作。即：如果该写操作对这个处理器来说完成了，那么这个处理器应该等待该写操作对所有处理器都完成了。
- 第三个条件保证了写操作的原子性。即读操作必须等待逻辑上先前的写操作变得全局可见

Understanding Atomicity – Caches 101



- A mechanism needed to propagate a write to other copies
- $\Rightarrow$ Cache coherence protocol

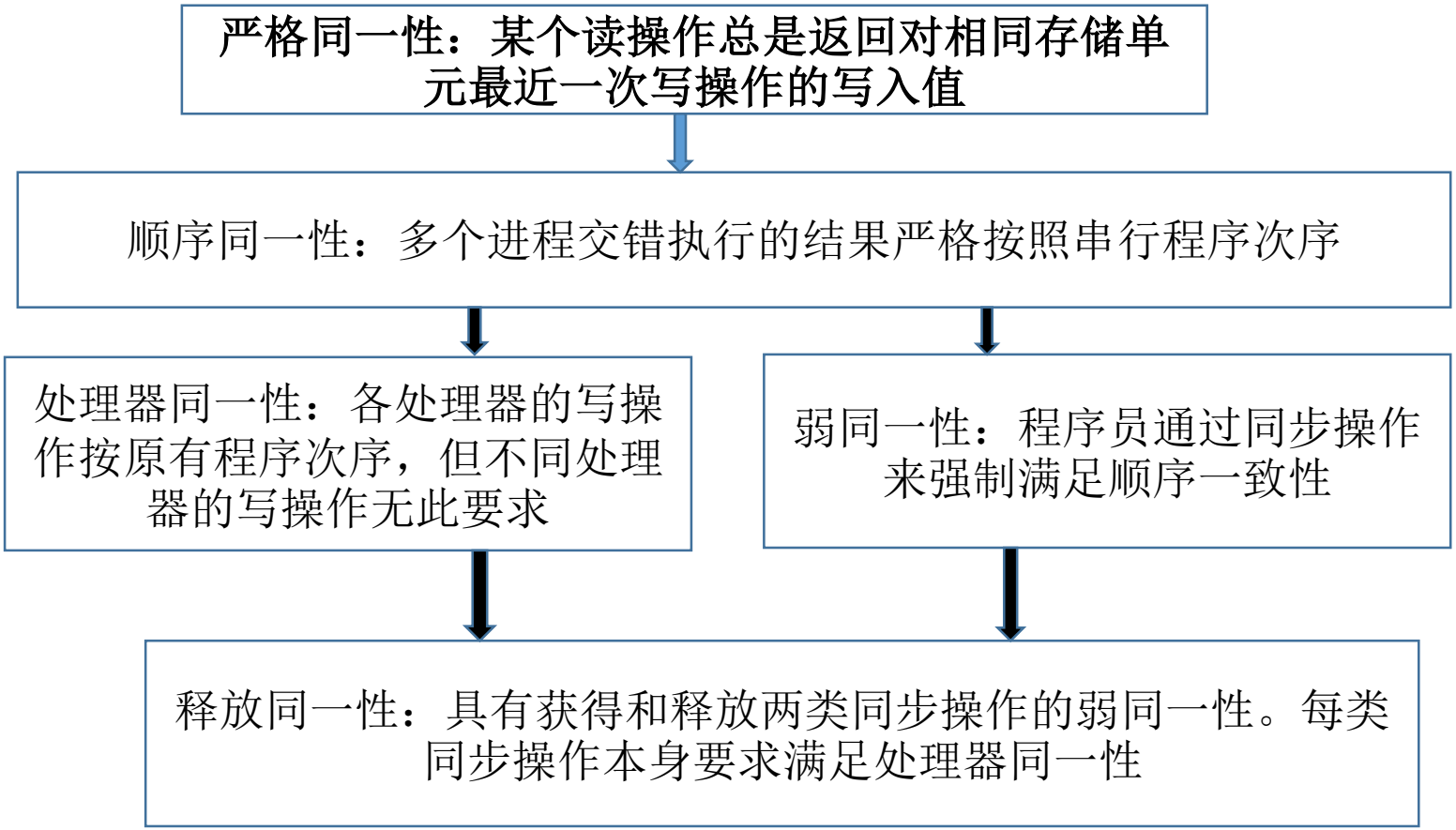
Sequential Consistency

- SC 约束了所有的存储器操作的序:
  - Write → Read
  - Write → Write
  - Read → Read, Write

- 是有关并行程序执行的简单模型
- 但是, 直觉上在单处理器上的合理的存储器操作的重排序违反SC模型
- 现代微处理器设计中一直都在应用重排序操作来获得性能提升(write buffers, overlapped writes, non-blocking reads…).

- Question: 如何协调性能提升与SC的约束？

Models of Memory Consistency



4种存储器同一性模型的直观定义

Relaxed Consistency Models

- Rules:
  - X → Y
    - Operation X must complete before operation Y is done
    - Sequential consistency requires:
      - R → W, R → R, W → R, W → W
  - Relax W → R
    - “Total store ordering”
  - Relax W → W
    - “Partial store order”
  - Relax R → W and R → R
    - “Weak ordering” and “release consistency”

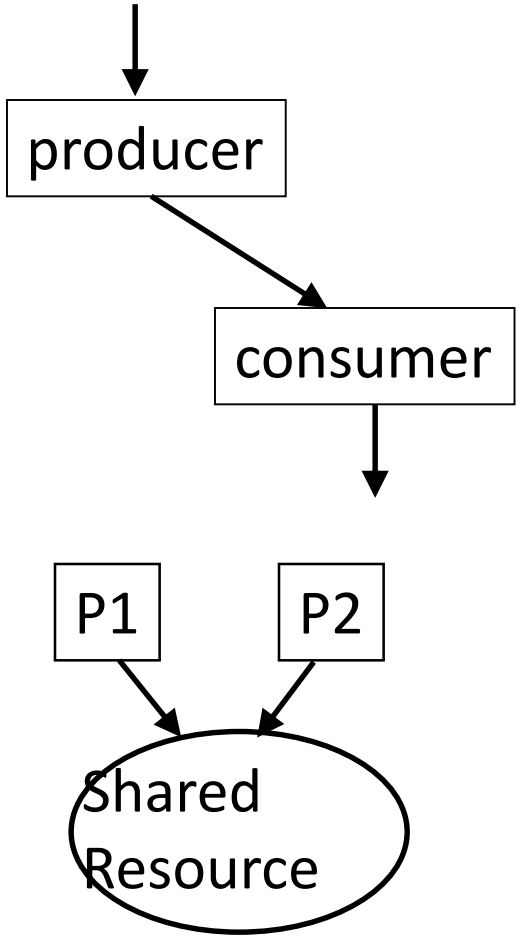
Synchronization

系统中只要存在并发进程，即使是单核系统都需要同步操作

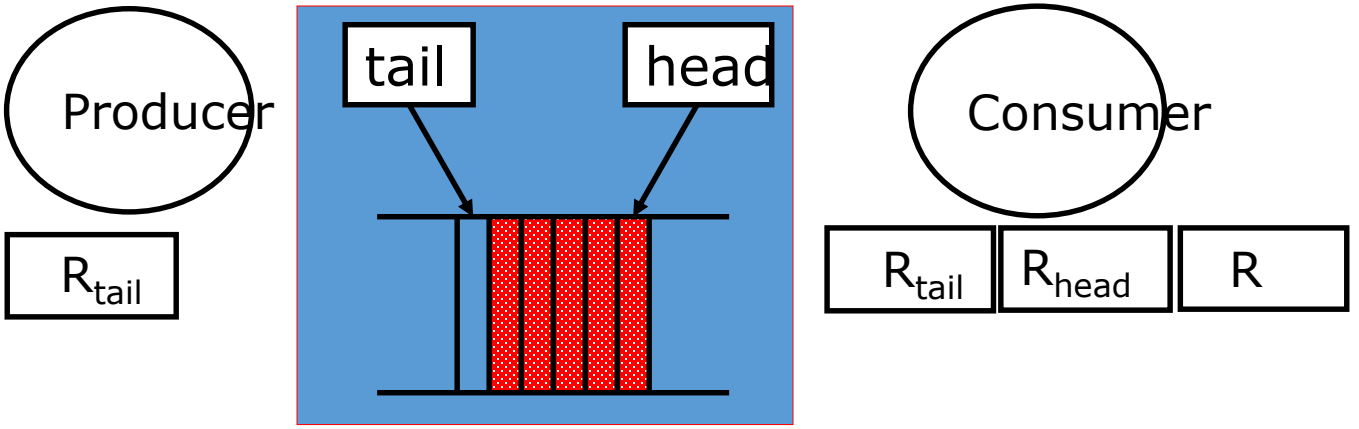
总体上存在两类同步操作问题:

*生产者-消费者问题：*一个 消费者进程必须等待生产者进程产生数据

互斥问题（Mutual Exclusion）: 保证在一个给定的时间内只有一个进程使用共享资源（临界区）



A Producer-Consumer Example



Producer posting Item x:

Load R_{tail}, (tail)

Store (R_{tail}), x

R_{tail}=R_{tail}+1

Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)

spin: Load R_{tail}, (tail)

if R_{head}==R_{tail} goto spin

Load R, (R_{head})

R_{head}=R_{head}+1

Store (head), R_{head}

process(R)

假设指令都是顺序执行的

Problems?

A Producer-Consumer Example

continued

Producer posting Item x:

Load R_{tail}, (tail)

1 Store (R_{tail}), x

R_{tail}=R_{tail}+1

2 Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)

spin: Load R_{tail}, (tail) *3*

if R_{head}==R_{tail} goto spin

Load R, (R_{head}) *4*

R_{head}=R_{head}+1

Store (head), R_{head}

process(R)

Can the tail pointer get updated before the item x is stored?

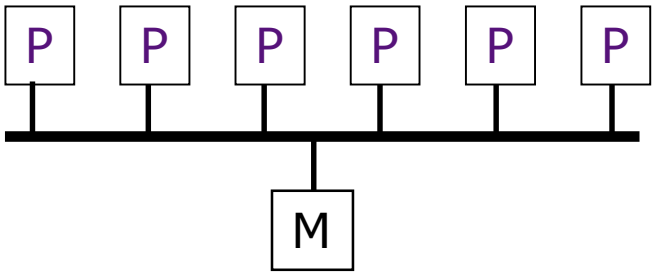
Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:

2, 3, 4, 1

4, 1, 2, 3

顺序同一性的存储器模型



“ A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

Leslie Lamport

Sequential Consistency =

多个进程之间的存储器操作可以任意交叉，但每个进程的存储器操作按照程序序

Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:
Store (X), 1 *(X = 1)*
Store (Y), 11 *(Y = 11)*

T2:
Load R_1, (Y)
Store (Y'), R_1 *(Y'= Y)*
Load R_2, (X)
Store (X'), R_2 *(X'= X)*

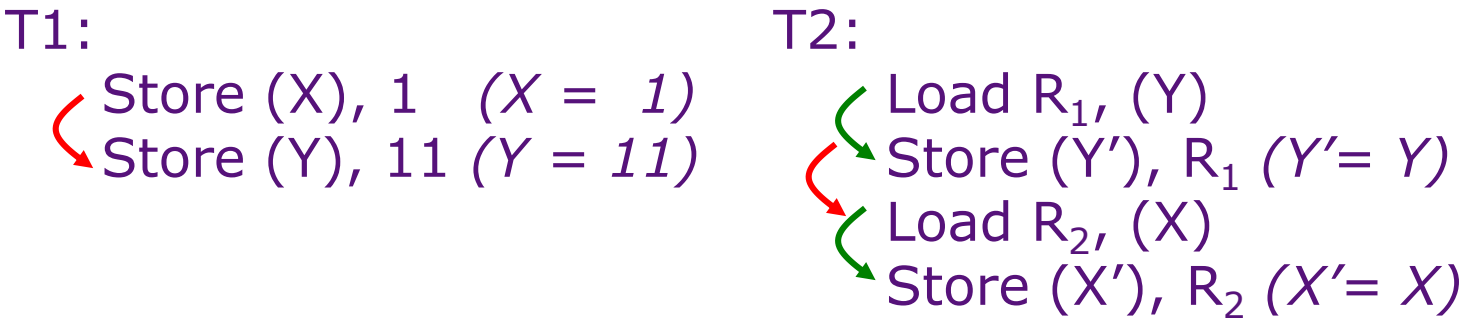
what are the legitimate answers for X' and Y' ?

(X',Y') ε {(1,11), (0,10), (1,10), ~~(0,11)~~} ?

If y is 11 then x cannot be 0

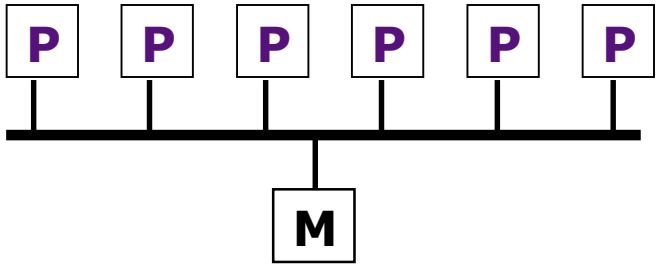
Sequential Consistency

存储同一性模型比单处理器上运行的程序的数据相关约束施加了更多的存储器操作约束



带有Cache 或 具有out-of-order执行能力的系统可以提供SC的存储器视图吗?

Issues in Implementing Sequential Consistency



现代计算机系统实现SC 的两个问题

- *Out-of-order execution capability*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a ≠ b
Store(a); Load(b)	*yes if* a ≠ b
Store(a); Store(b)	*yes if* a ≠ b

- *Caches*

  Cache使得某一处理器的store操作不能被另一处理器即时看到

No common commercial architecture has a sequentially consistent memory model!

Memory Fences

Instructions to sequentialize memory accesses

实现弱同一性或放松的存储器模型的处理器（允许针对不同地址的loads 和stores操作乱序）需要提供存储器栅栏指令来强制对某些存储器操作串行化

Examples of processors with relaxed memory models:

Sparc V8 (TSO,PSO): Membar

Sparc V9 (RMO):

Membar #LoadLoad, Membar #LoadStore

Membar #StoreLoad, Membar #StoreStore

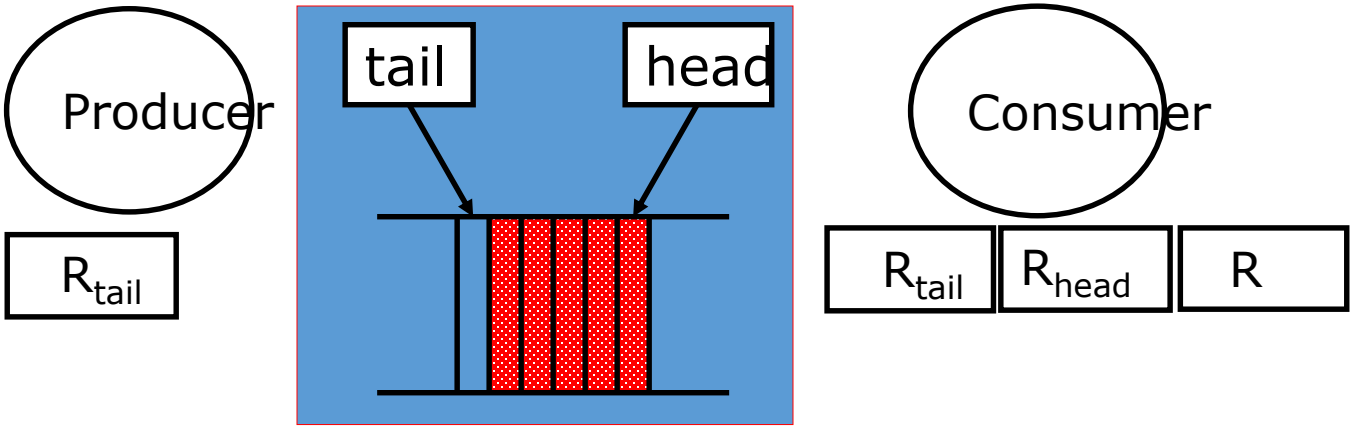
PowerPC (WO): Sync, EIEIO

ARM: DMB (Data Memory Barrier)

X86/64: mfence (Global Memory Barrier)

存储器栅栏是一种代价比较大的操作，仅仅在需要时，对存储器操作串行化

Using Memory Fences



Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
Membar$_{SS}$
R_{tail}=R_{tail}+1
Store (tail), R_{tail}

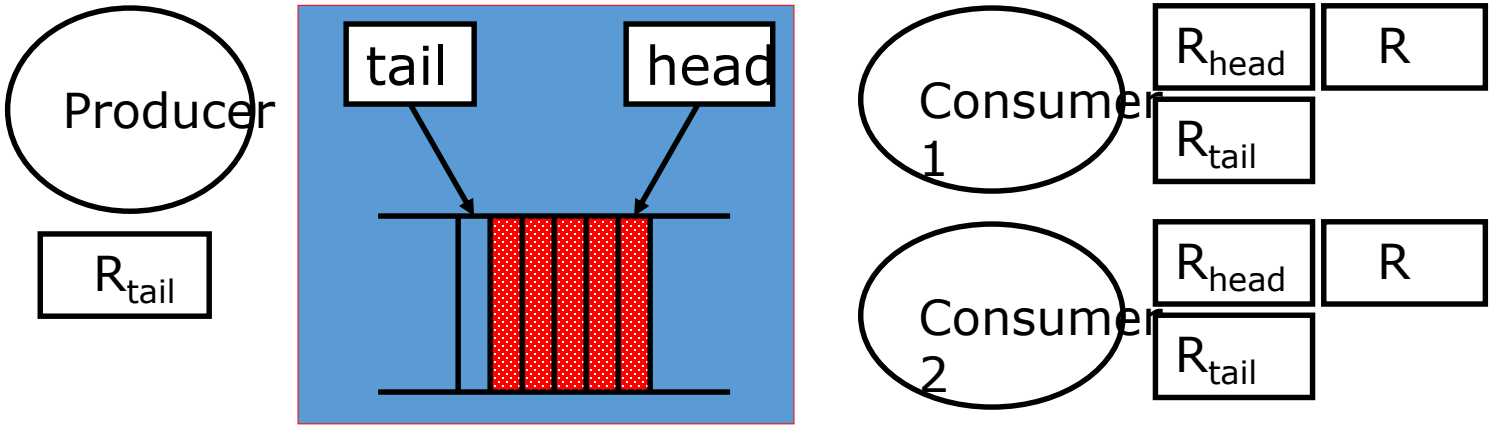
ensures that tail ptr is not updated before x has been stored

Consumer:

Load R_{head}, (head)
spin: Load R_{tail}, (tail)
if R_{head}==R_{tail} goto spin
Membar$_{LL}$
Load R, (R_{head})
R_{head}=R_{head}+1
Store (head), R_{head}
process(R)

ensures that R is not loaded before x has been stored

Multiple Consumer Example



Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
R_{tail}=R_{tail}+1
Store (tail), R_{tail}

Consumer:

spin: Load R_{head}, (head)
Load R_{tail}, (tail)
if R_{head}==R_{tail} goto spin
Load R, (R_{head})
R_{head}=R_{head}+1
Store (head), R_{head}
process(R)

Critical section:
Needs to be executed atomically by one consumer

What is wrong with this code?

Mutual Exclusion Using Load/Store

基于两个共享变量c1和c2的同步协议。初始状态c1和c2均为0

Process 1

```
     ...
     c1=1;
L:   if c2=1 then go to L
       < critical section>
     c1=0;
```

Process 2

```
     ...
     c2=1;
L:   if c1=1 then go to L
       < critical section>
     c2=0;
```

What is wrong? *Deadlock!*

Mutual Exclusion: *second attempt*

为避免死锁，我们让一进程等待时放弃reservation（预订）
(i.e. Process 1 sets c1 to 0).

Process 1

```
    ...
L:  c1=1;
    if c2=1 then
        { c1=0; go to L}
      < critical section>
    c1=0
```

Process 2

```
    ...
L:  c2=1;
    if c1=1 then
        { c2=0; go to L}
      < critical section>
    c2=0
```

- 死锁显然是没有了，但有可能会发生 活锁 现象
- 可能还会出现某个进程始终无法进入临界区 ⇒ *starvation*

A Protocol for Mutual Exclusion

T. Dekker, 1966

基于3个共享变量c1, c2 和turn的互斥协议，初始状态三个变量均为0.

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

- turn = *i* *保证仅仅进程 i 等待*
- 变量 c1 and c2 保证n个进程互斥地访问临界区，该算法由*Dijkstra给出，相当精巧*

Locks or Semaphores

E. W. Dijkstra, 1965

信号量（A *semaphore*） 是一非负整数, 具有如下操作:

P(s): *if s>0, decrement s by 1, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P's and V's 必须是原子操作, i.e., 不能被中断，不能由多个处理器交叉访问s

```
Process i
    P(s)
        <critical section>
    V(s)
```

s的初始值设置为可访问临界区的最大进程数

Implementation of Semaphores

在顺序同一性模型中，信号量 (mutual exclusion) 可以用常规的 Load and Store 指令实现，但是互斥的协议很难设计

一种简单的解决方案是提供:

atomic read-modify-write instructions

Examples: *m is a memory location, R is a register*

Test&Set (m), R:
    R ← M[m];
    if R==0 *then*
        M[m] ← 1;

Fetch&Add (m), R_V, R:
    R ← M[m];
    M[m] ← R + R_V;

Swap (m), R:
    R_t ← M[m];
    M[m] ← R;
    R ← R_t;

Multiple Consumers Example

using the Test&Set Instruction

```
P:      Test&Set (mutex),Rtemp
        if (Rtemp!=0) goto P
        Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rhead=Rhead+1
        Store (head), Rhead
V:      Store (mutex),0
        process(R)
```

Critical Section

其他原子的read-modify-write 指令 (Swap, Fetch&Add, etc.) 也能实现 P's and V's操作

Nonblocking Synchronization

```
Compare&Swap(m), Rt, Rs:
  if (Rt==M[m])
     then  M[m]=Rs;
           Rs=Rt ;
           status ← success;
     else  status ← fail;
```

status is an *implicit argument*

```
try:
spin:  Load Rhead, (head)
       Load Rtail, (tail)
       if Rhead==Rtail goto spin
       Load R, (Rhead)
       Rnewhead = Rhead+1
       Compare&Swap(head), Rhead, Rnewhead
       if (status==fail) goto try
       process(R)
```

Performance of Locks

Blocking atomic read-modify-write instructions

e.g., Test&Set, Fetch&Add, Swap

vs

Non-blocking atomic read-modify-write instructions

e.g., Compare&Swap,

Load-reserve/Store-conditional

vs

Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:

degree of contention,

caches,

out-of-order execution of Loads and Stores

later ...

Acknowledgements

- These slides contain material developed and copyright by:
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - David Patterson (UCB)
  - Chenxi Zhang (Tongji)
  - Sarita Adve (UIUC)
  - Muhamed Mudawar (..)
- UCB material derived from course CS152、CS252