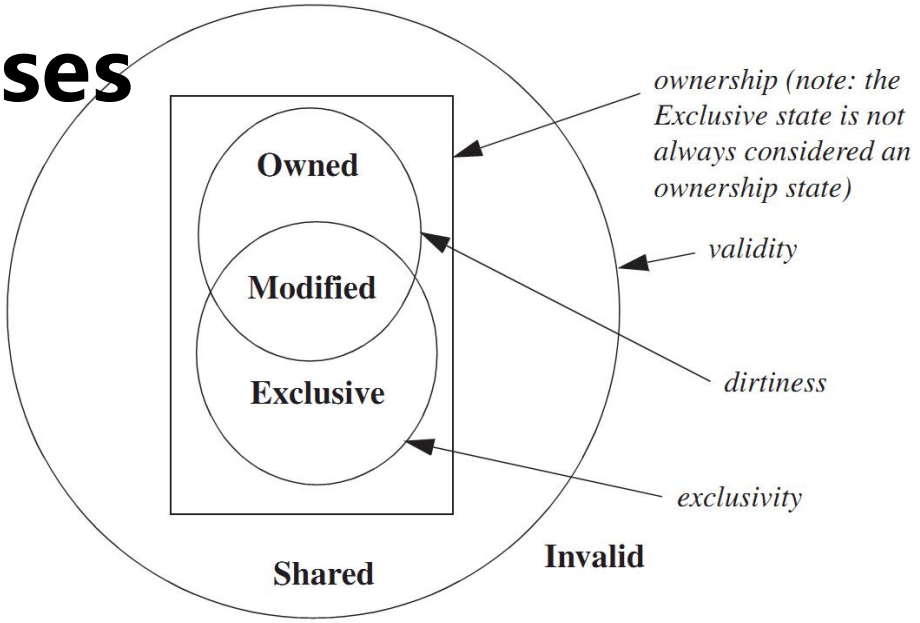
计算机体系结构

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

review

- **共享数据块的跟踪：监听和目录**
- **Cache一致性协议实现：写作废和写更新**
- **集中式共享存储 Cache一致性协议**
  - Snooping协议：MSI，MESI，MOESI
- **Coherency Misses**
  - True Sharing
  - False Sharing



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 存储一致性

7.5 同步与通信

7.3 分布式共享存储器体系结构

01 **问题分析**

02 基于目录的一致性协议

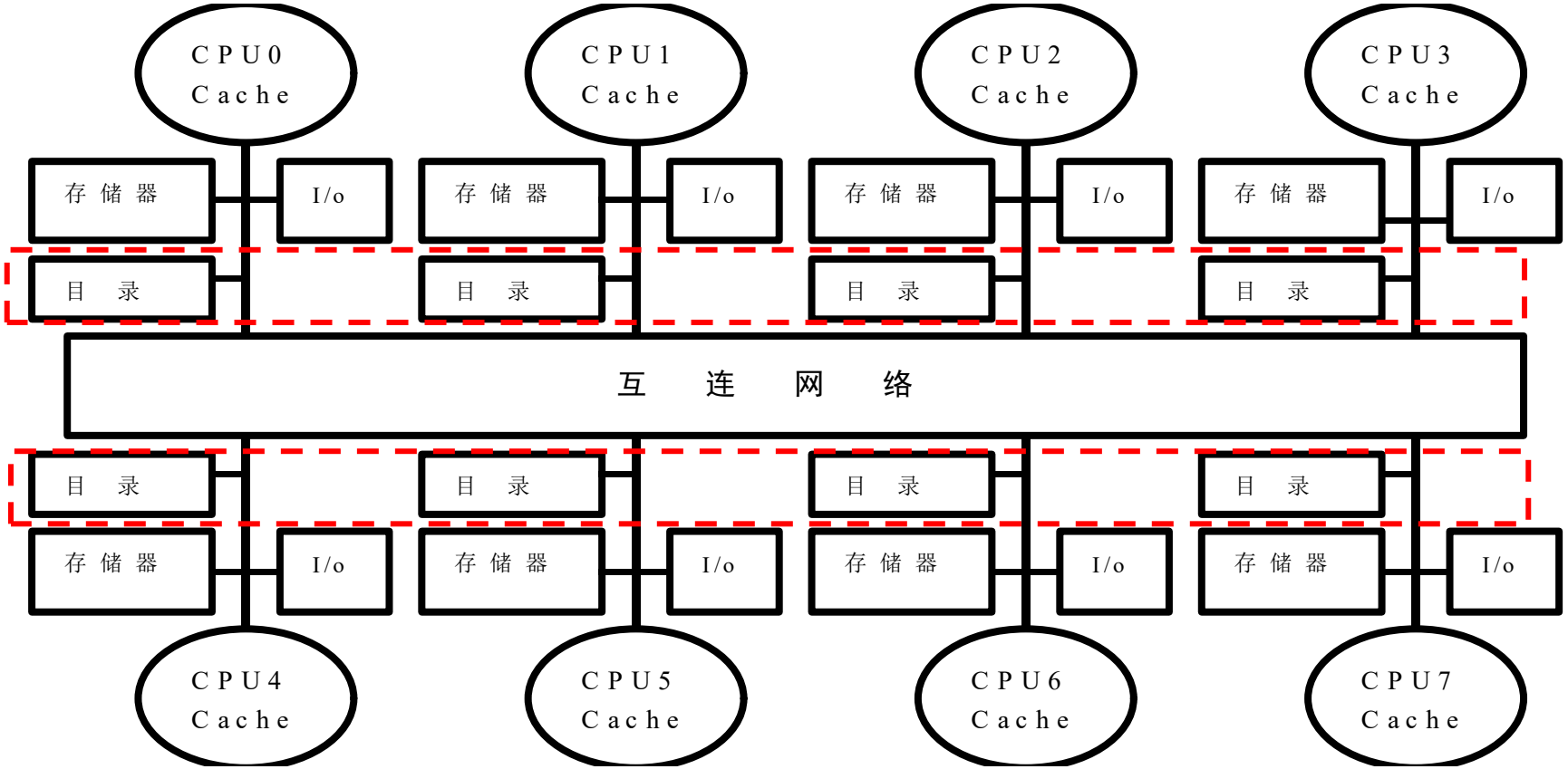
Limitations of Snooping Protocols

- **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
- **解决方案之一：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
  - 可扩放性仍然受到限制。
- **在非总线或环的网络上监听是比较困难的**
  - 将一致性相关信息**广播**到所有处理器，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。

分布式共享存储器体系结构



- **对每个结点增加目录表后的分布式存储器的系统结构，如上图**
- **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**

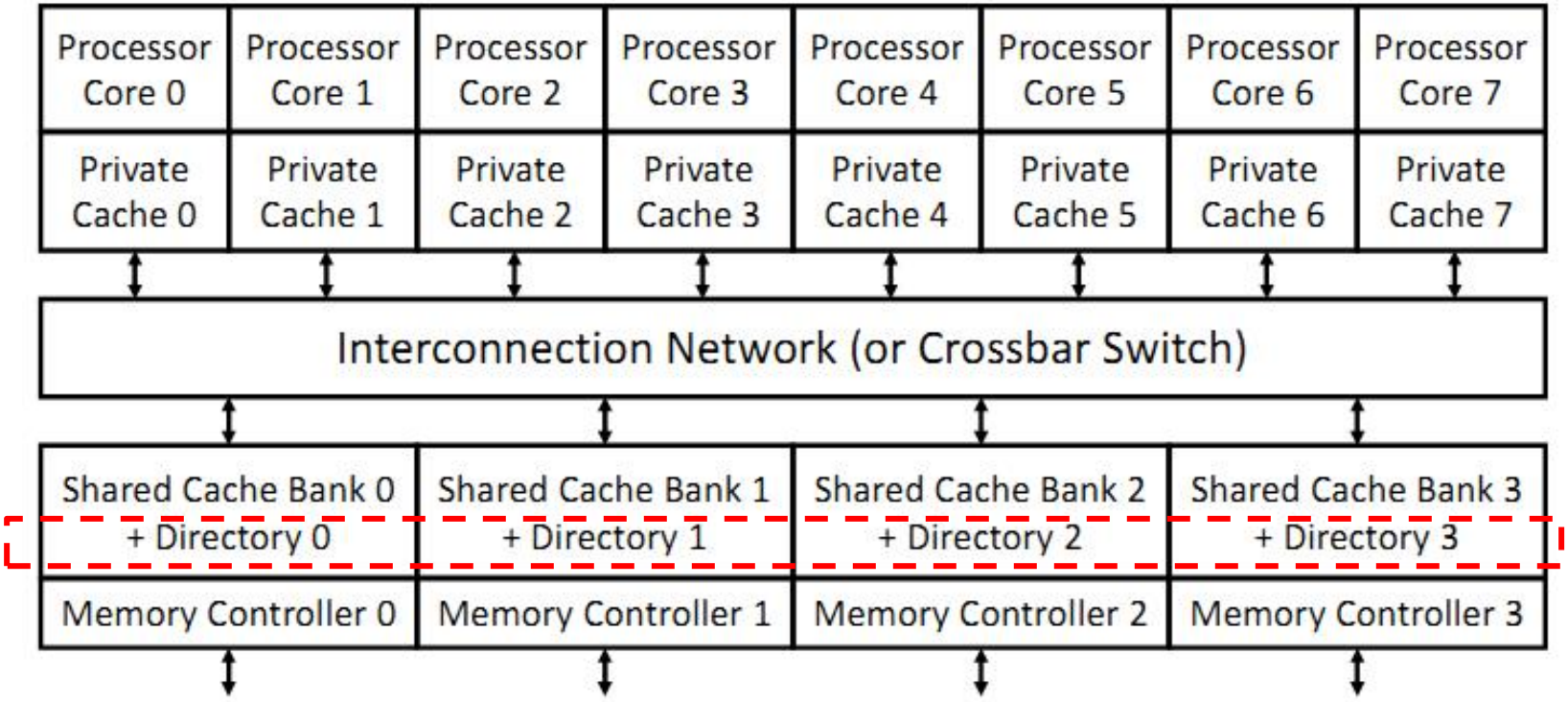
7.3 分布式共享存储器体系结构

01 问题分析

02 **基于目录的一致性协议**

Directory in a Chip Multiprocessor

- **目录在所有处理器共享的最外层Cache（共享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
  - 在私有Cache中块的状态： State = M (Modified), S (Shared), or I (Invalid)

Block in a Private Cache

Block in a Shared Cache

一些术语

- **本地或私有Cache （Local / 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，不用广播方式到所有处理器核**
  - 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

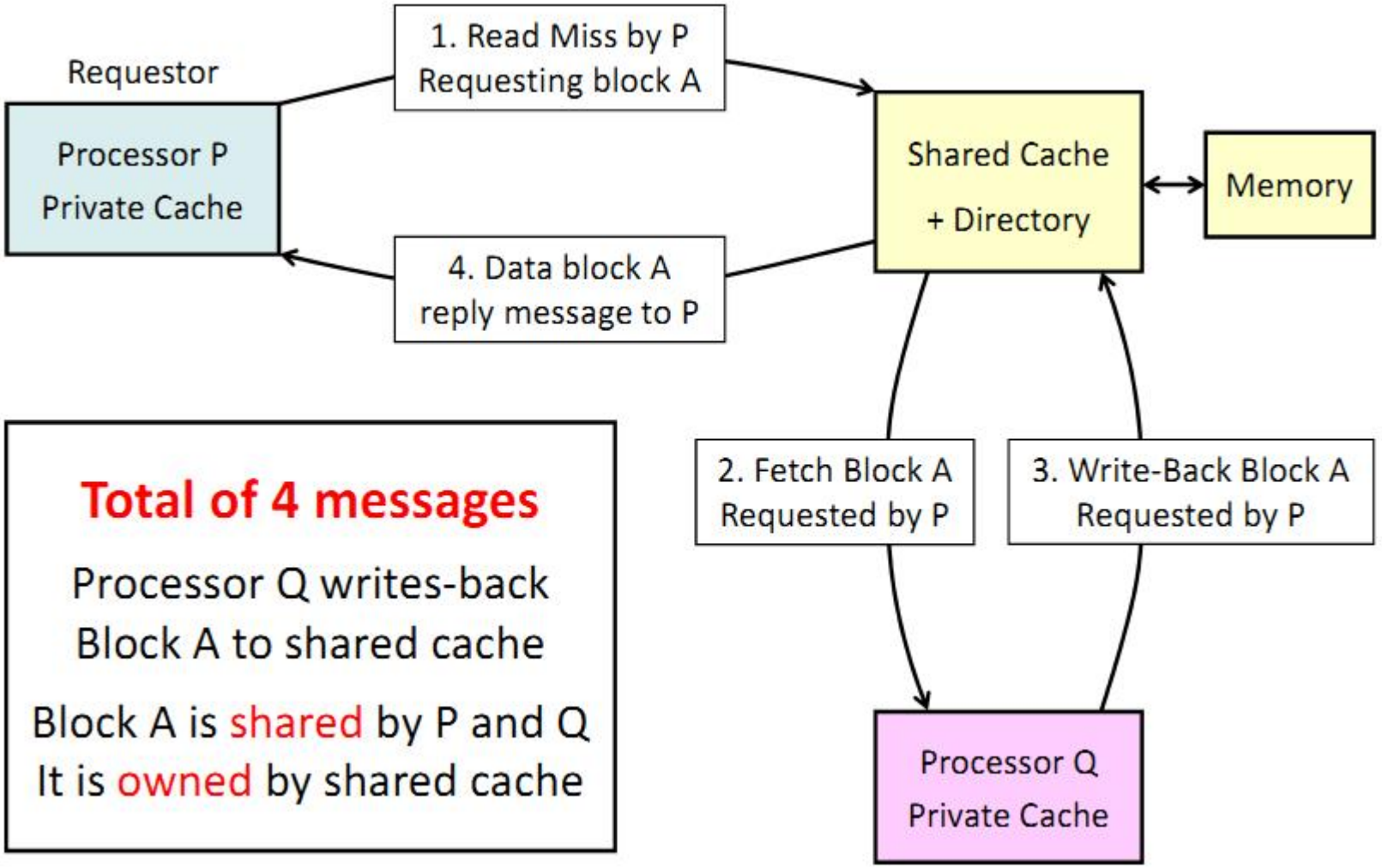
3. Shared: 该块可能被复制到多个cache中

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** 给拥有该块的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, 并将对应于P的presence bit置位
  - 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P，并将对应 P的presence bit置位
  - 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**

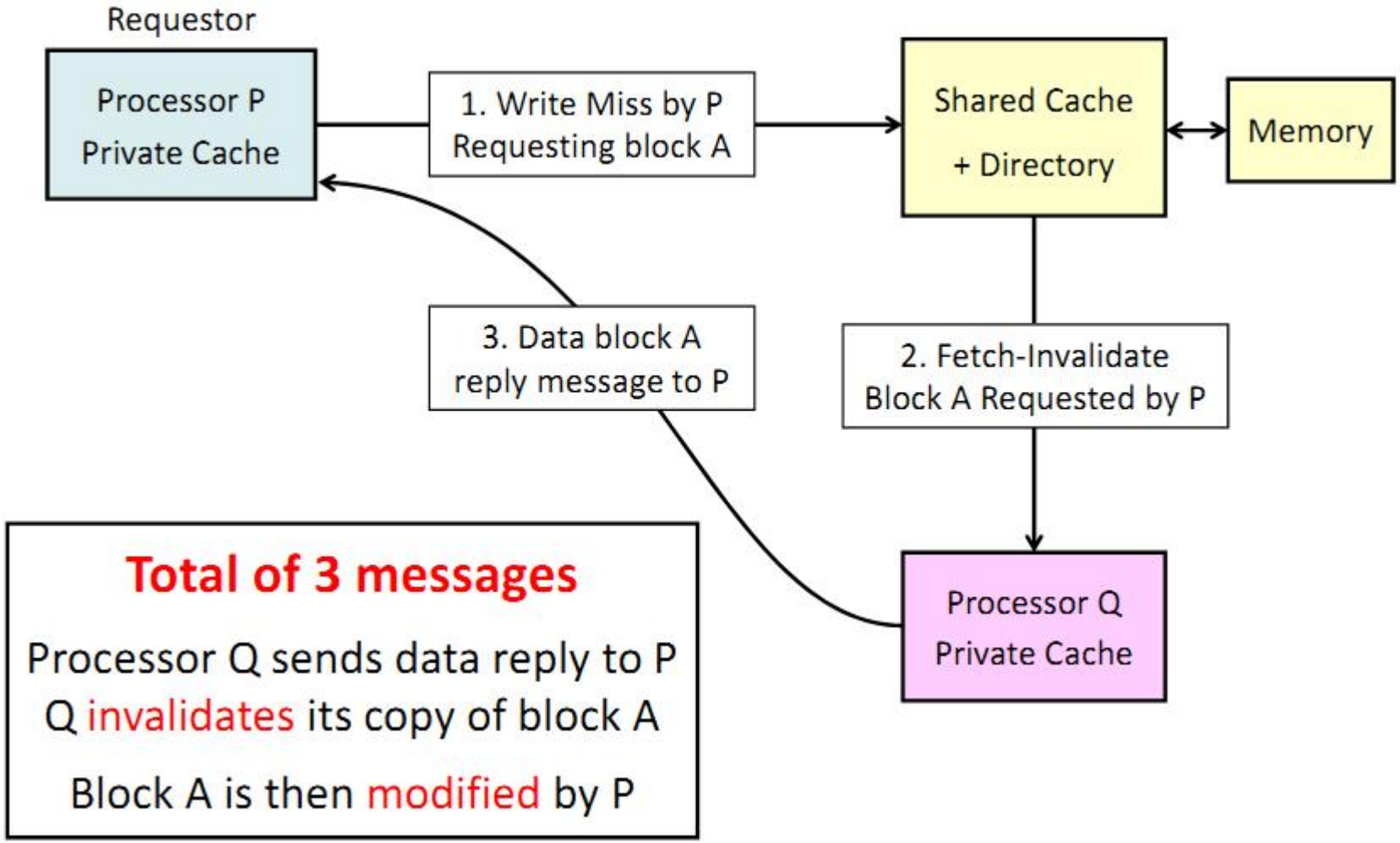
Read Miss to a Block in Modified State



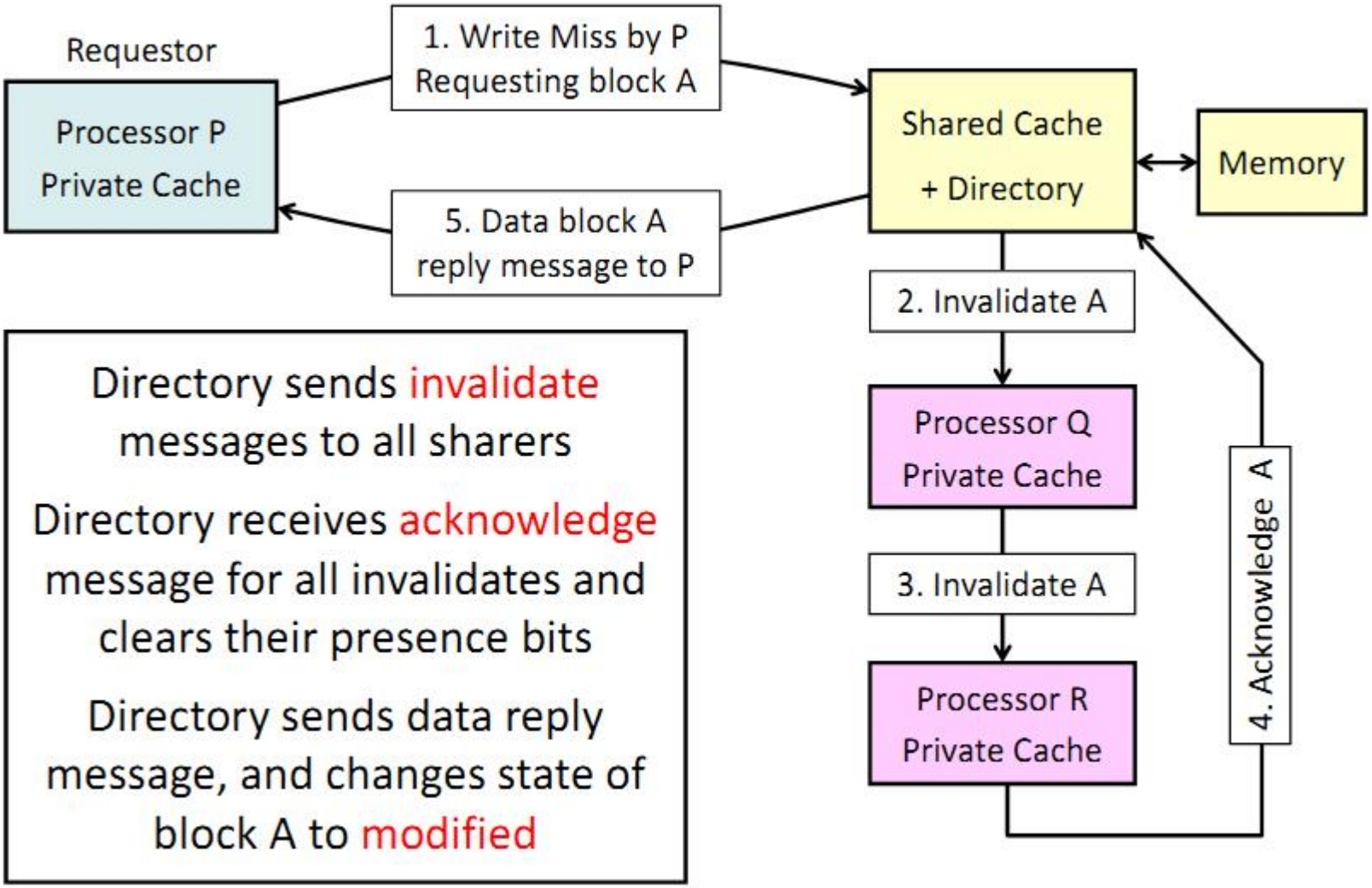
Write Miss Message by P to Directory

- **Home Directory: block 是Modified态**
  - Directory 发送 **Fetch-Invalidate message** 给处理器Q的Cache (Remote Cache) （Q包含该块的最新值）
  - 处理器Q的cache 直接发送数据应答消息给P
  - Q的cache将对应块的状态修改为invalid
  - P的cache (Local) 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**

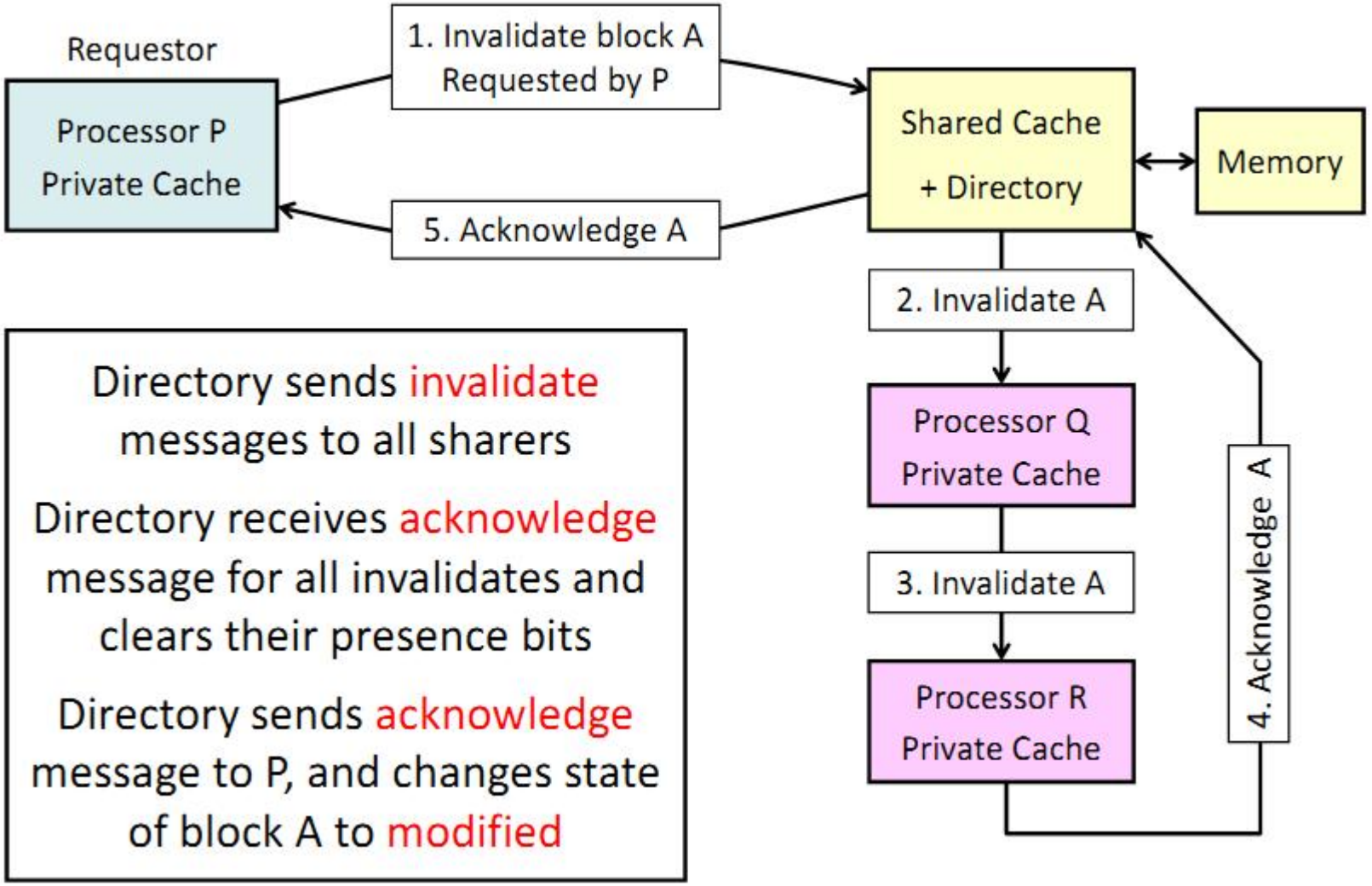
Write Miss to a Block in Modified State



Write Miss to a Block with Sharers



Invalidating a Block with Sharers

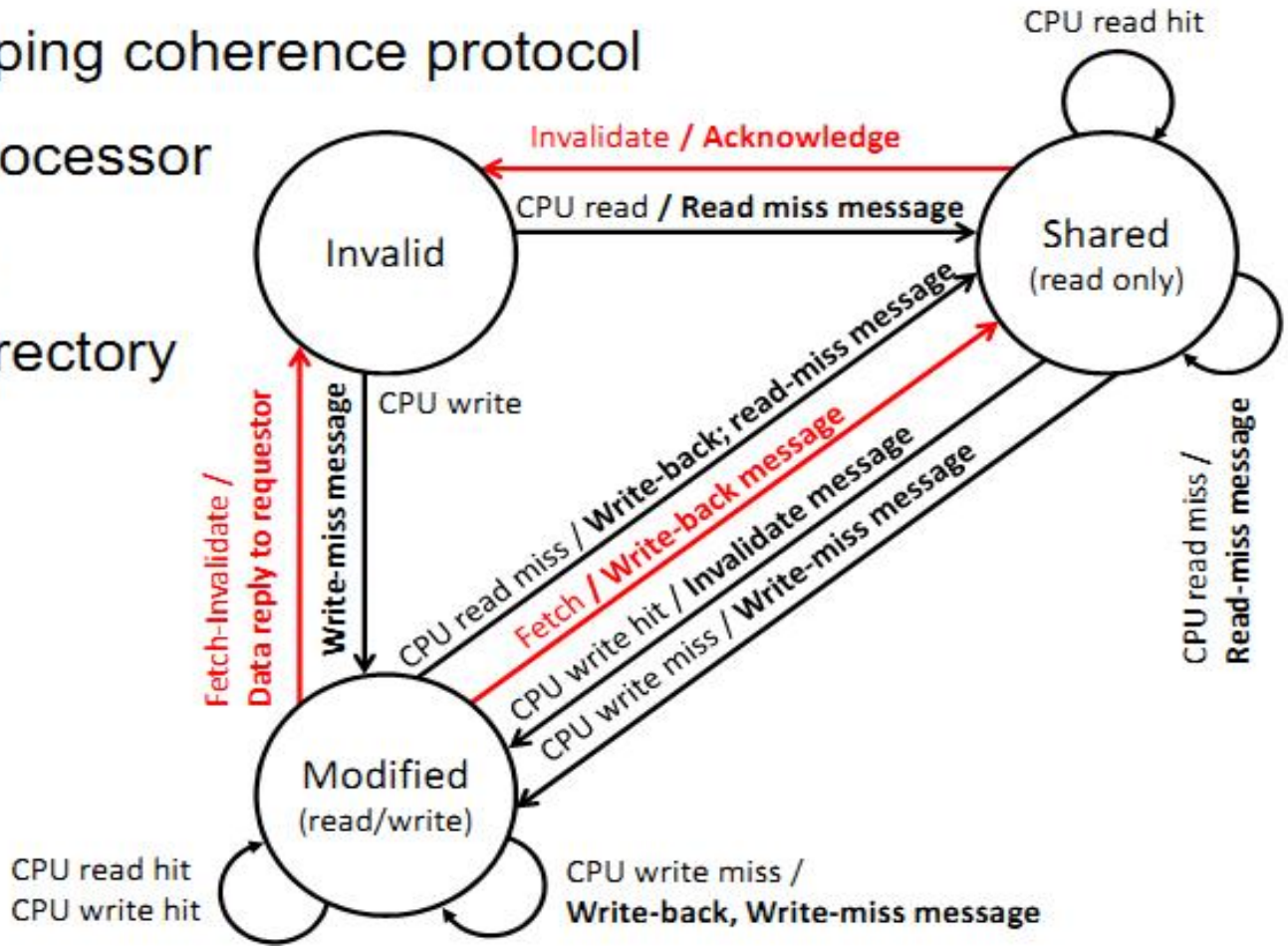


Directory Protocol Messages

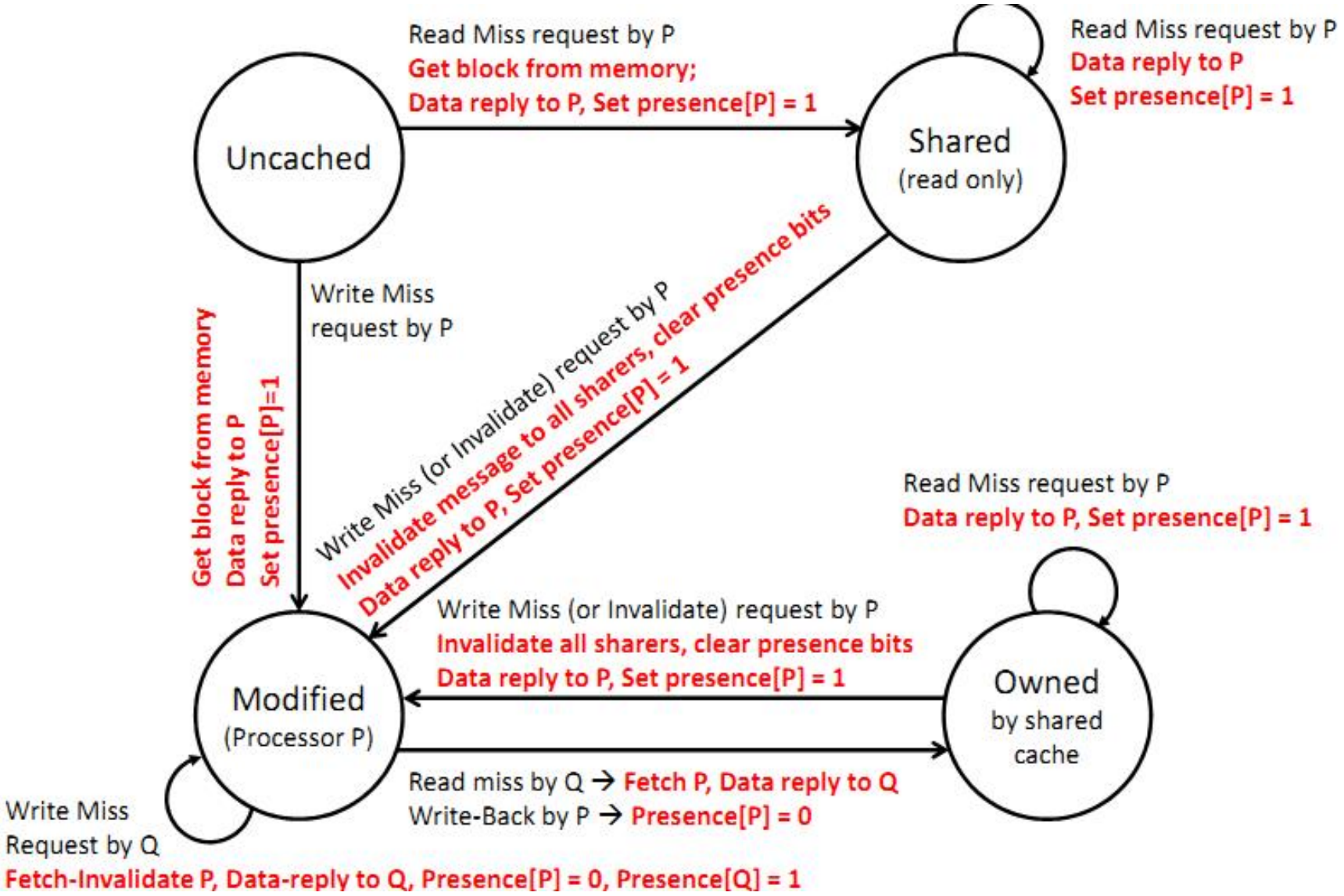
Message Type	Source	Destination	Message Function
Read Miss	Local Cache	Home Directory	Processor P has a read miss at address A Request data and make P a read sharer
Write Miss	Local Cache	Home Directory	Processor P has a write miss at address A Request data and make P the exclusive owner
Invalidate	Local 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**

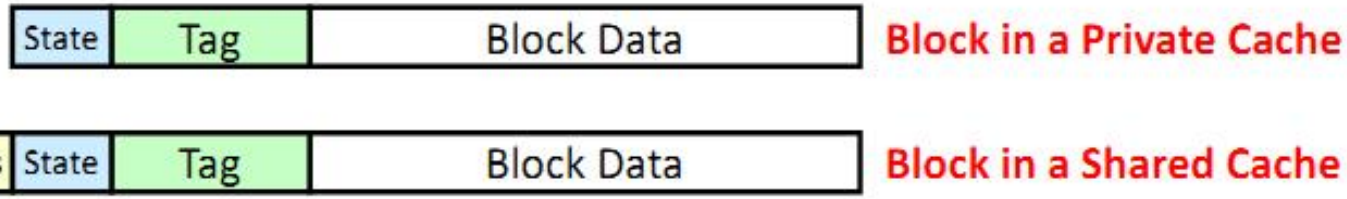


MOSI State Diagram for Directory



Summary

- **分布式共享存储的Cache一致性协议**
  - Cache块的状态：
    - 私有Cache中块的状态 /目录中Cache块的状态



  - 状态迁移过程：状态迁移图
- **存储一致性问题**
  - Consistency研究不同处理器访问存储器操作的定序问题，即所有处理器发出的所有访问存储器操作(所有地址）的**全序**
  - Coherence研究不同处理器访问存储器相同地址操作的定序问题，即访问每个Cache块的**局部序**问题

7.4 存储一致性问题

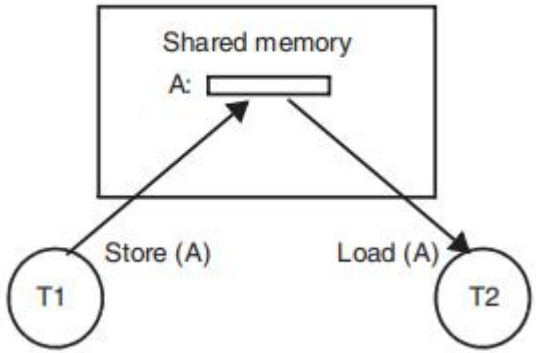
01 什么是存储一致性

02 顺序存储一致性

03 放松的存储一致性

recap：共享内存通信

- **不同线程通过Load/Store操作完成值的“隐式”通信**
  - T1：生产者线程；T2：消费者线程
  - 程序员期望的顺序：Store→Load
  - 如果没有同步操作，结果会怎样?
- **程序员通过编写代码来实现同步**
  - 程序员希望的过程：P1：Store A →Store flag；
  - P2观察到的这两次更新的顺序是一样的。即：
  - P2 观察flag = 1，那么A=1
  - 如果P2观察到的这两次更新顺序相反，则A = ?
- **Cache一致性通过写传播，使Store的值传播到所有多份副本**
  - 它给上层软件一个错觉：每个内存位置只有一份单一副本（实际上可能有多个副本）



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

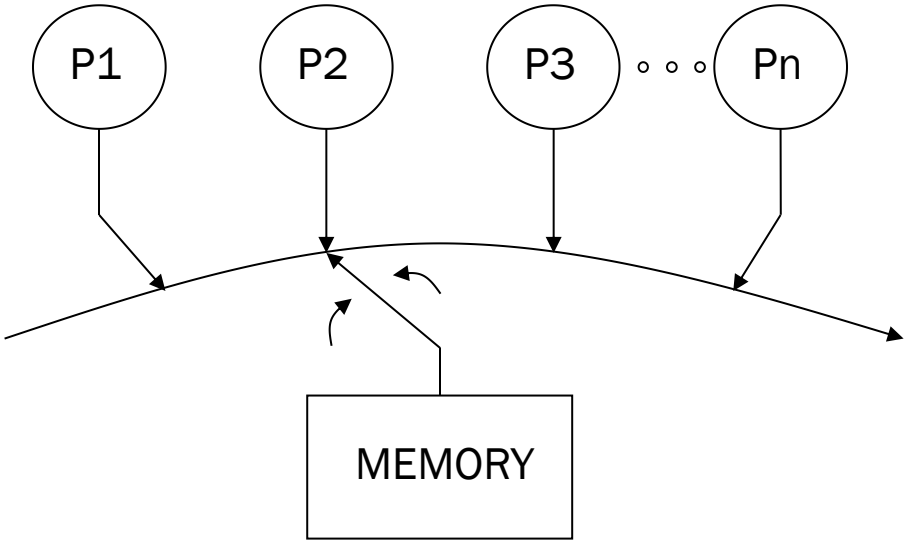
存储一致性的定义

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

- **Cache 一致性**
  - 保证的是当修改共享存储空间中的某一单元后，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”。
  - 但并不要求Store值立即传播，也不要求对应内存位置的所有副本必须在任何时候都一样
    - 基于总线的MSI-写作废协议中，新值是在“**总线被释放的那一刻，对所有处理器可见**“
  - 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
    - 如上例，P2可能先感知到flag的更新，导致P2感知的P1的更新顺序与P1的程序序相反
  - 没有涉及处理器P1和P2对不同地址单元的访问顺序
- **存储一致性（Memory Consistency）模型定义: 共享地址空间的存储一致性模型是在多个处理器对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。**

Implicit Memory Model

- **顺序一致性(Sequential Consistency) [Lamport]：该模型要求所有处理器的读、写和交换(swap)操作以某种序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： 不论指令流如何交叠执行，全局序必须保持所有进程的程序序**
  - 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
  - 不同进程的指令交错执行



- **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)
P1:r0=r0+1
P1:Write(r0,X)
P2:r1=r1+1
P2:Write(r1,X)
    x=3
```

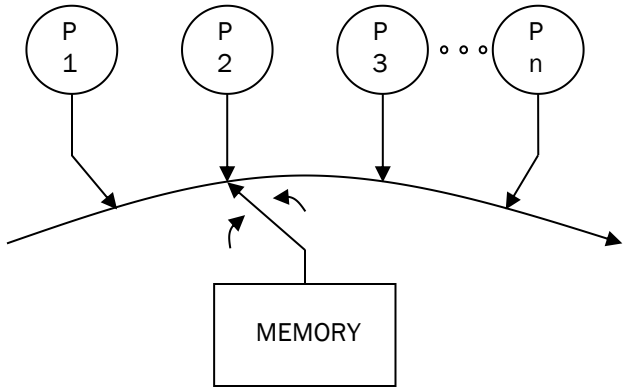
```
P2:r1=Read(X)
P2:r1=r1+1
P2:Write(r1,X)
P1:r0=Read(X)
P1:r0=r0+1
P1:Write(r0,X)
   x=4
```

不同进程的指令交错执行
交错方式不同，结果不同

是否满足Sequential Consistency?

Understanding Program order-Example 2

P1	P2	P3
A=1;	while (A==0);	while (B==0);
	B = 1;	print A;



No caches, no write buffers

假设A,B的初始值为0；

从程序员角度看：P3应该输出 A=1；

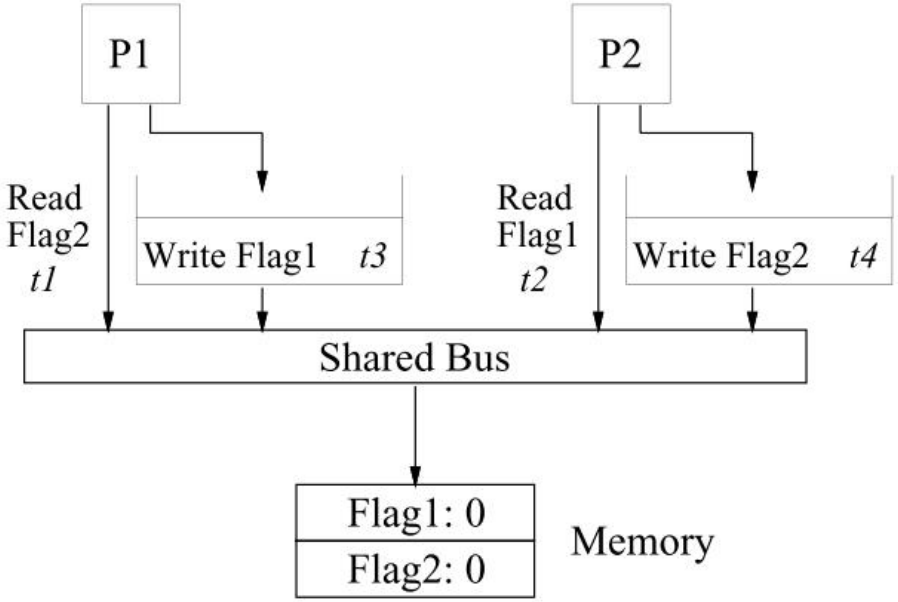
如果程序序和存储器访问序一致，则P3 输出与预期相同

如果P2被允许越过对变量A的读操作（程序序与存储器访问序不一致），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。

但是上面的模型导致体系结构的许多优化策略难以实施

假如我们做一些性能优化工作，结果会怎样?

Optimization 1: Write Buffers with Bypassing Capability

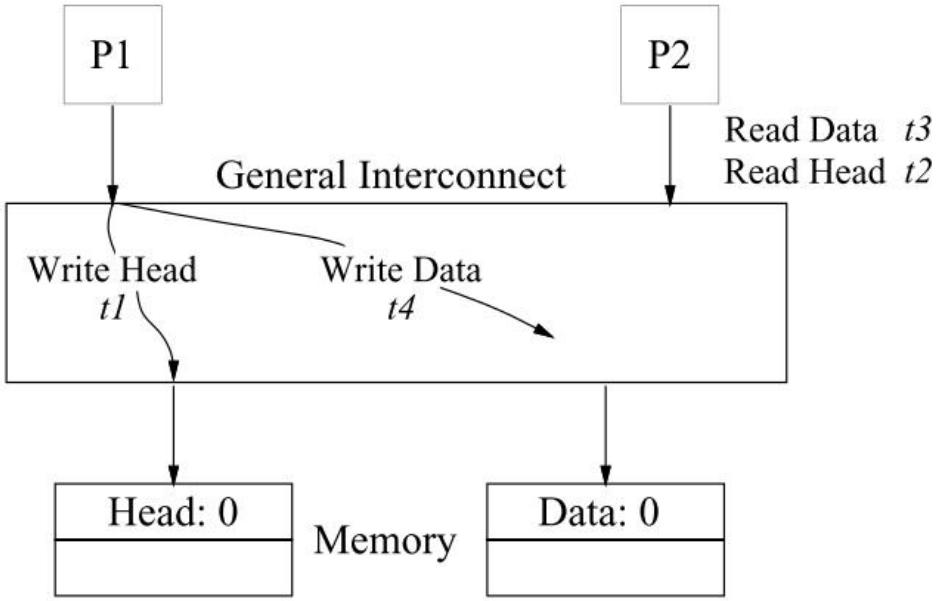


P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(a) write buffer

- **ti：表示实际完成的读写序**
- **Flag1和Flag2的新值都在write buffer中**
- **导致存储器操作的序与程序序不同，从全局看违反SC规则，P1和P2可同时进入临界区**

Optimization 2: Overlapping Write Operations

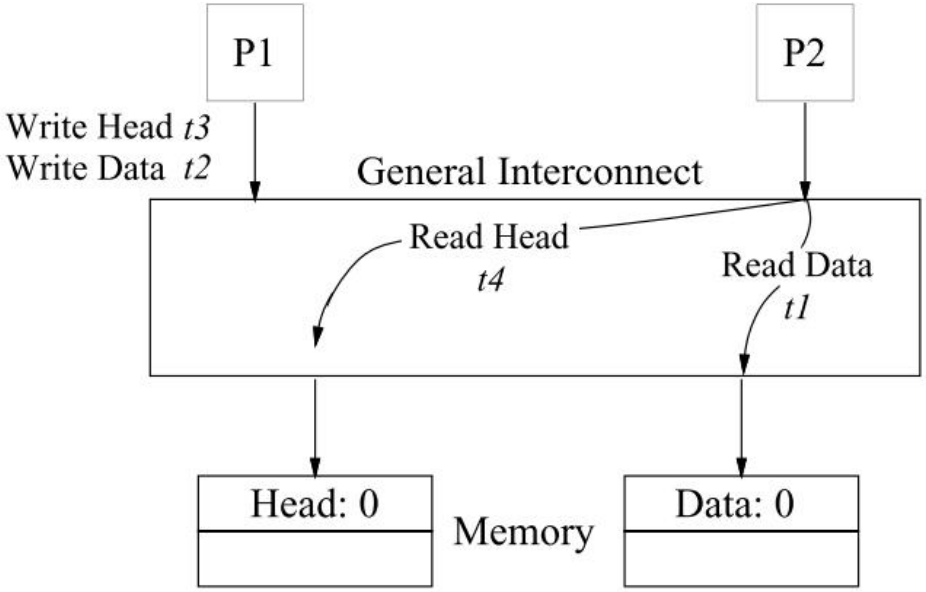


P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(b) overlapped writes

- **非总线互联网络：避免总线的性能瓶颈**
- **多存储器模块：具有并行读写特性，提高读写性能**
  - 导致write Data 与 Write Head的完成序 与 程序序 相反
  - 进而 导致 P2 首先**读到Head的新值，Data 的旧值**，违反SC 规则

Optimization 3: Non-blocking reads



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(c) non−blocking reads

- **假设P1写操作按照程序序执行存储器操作，P2允许以overlapped 的方式执行读操作 （non-blocking read, speculative execution, and dynamic scheduling)**
- **则：可能导致P2 Read Data 提前于 P1的Write Data的情况，导致违反SC规则**

多处理器存储器操作的困难

- **大多数并行计算机体系结构所研究的问题：**
  - 如何克服顺序执行和并行执行的瓶颈，以得到更高的性能和效率
  - 如何为用户提供良好的编程模型，以便编写正确而高性能的并行程序
- **Load/store操作的顺序问题**
  - Operations: A, B, C, D
    - 硬件以何种顺序执行（和报告结果）这些操作?
  - 程序员与微结构设计人员的协议由ISA来约定
  - 保留程序员所希望的执行顺序
    - 可降低编程的难度，如：易于 debugging; 易于状态恢复、异常处理等
    - 通常会使得硬件设计变得困难，特别是当我们的设计目标为高性能处理器时，乱序load-store的执行，使得问题变得复杂

单个处理器存储器操作的序

- **操作顺序由von Neumann 模型约定**
- **顺序串行执行**
  - 硬件执行load和store操作以程序序顺序执行
- **乱序执行不改变程序语义**
  - 硬件以程序序报告load和store操作的结果
- **优点:**
  - 在执行时机器状态是确定的
  - 程序重复运行机器状态是一致的，有利于程序调试
- **缺点: 维护这种序的额外开销, 降低了性能,增加了复杂性, 降低了可扩放性**

数据流处理器的存储器操作的序

- 当操作数准备好就可以执行存储器操作
- 操作的顺序仅仅由数据依赖性来确定
- 相互独立的操作可以以任意序执行和提交结果

- 优点: 并行度高，性能高
- 缺点: 相同程序的不同次运行次序可以不同，使得调试困难

MIMD处理器中的存储器操作序

- 每个处理器的存储器操作以顺序序执行对应于运行在该处理器上的一个线程 (假设每个处理器符合von Neumann模型)
- 多个处理器并发地执行存储器操作
- 存储器如何看来自所有处理器的存储器操作?
  - 即不同处理器发出的存储器操作,在共享存储器端看到的应该是什么序?

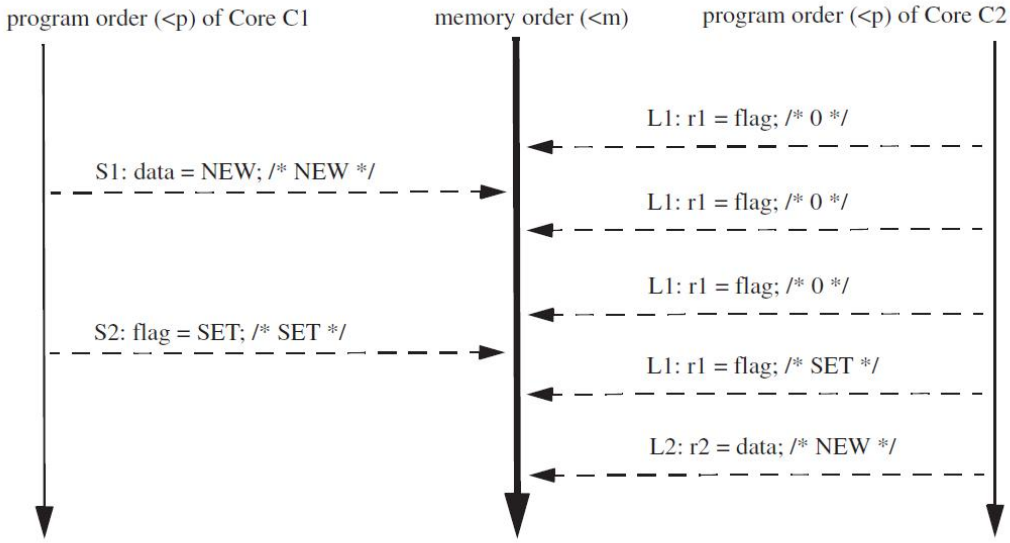
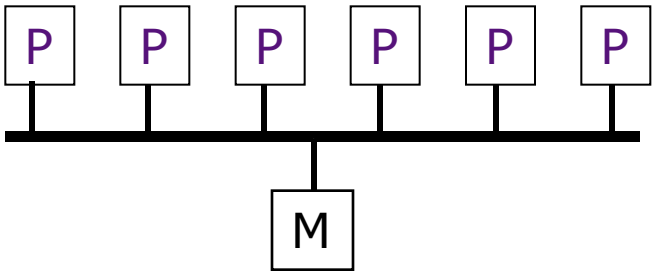


FIGURE 3.1: A Sequentially Consistent Execution of Table 3.1's Program.

MIMD处理器中序的重要性

- **从易于调试角度看：**
  - 若程序的每次执行都是相同的序有利于程序的调试
- **从维护正确性角度看：**
  - 若从不同处理器看到的存储器操作序不同，增加了维护正确性的难度
- **性能和代价权衡**
  - 强制符合严格 “sequential ordering”使得硬件设计人员实现性能优化技术变得十分复杂 (例如., OoO 执行, caches)
- **<p ： 程序序(program order)**
  <m ： 存储器操作序(memory order)

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

每个进程的存储器操作按照程序序

顺序一致性的充分条件

- **多个进程可以交织执行，但顺序一致性模型没有定义具体的交织方式，满足每个进程程序序的总体执行序可能会很多。因此有下列定义：**
  - 顺序一致性的执行：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序产生的结果一致，那么程序的这次执行就称为是顺序一致的。
  - 顺序一致性的系统：如果在一个系统上的任何可能的执行都是顺序一致的，那么这个系统就是顺序一致的

顺序一致性的充分条件

- 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存储操作
- 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**全局完成**，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

顺序一致性示例

TABLE 3.1: Should r2 Always be Set to NEW?		
Core C1	**Core C2**	**Comments**
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

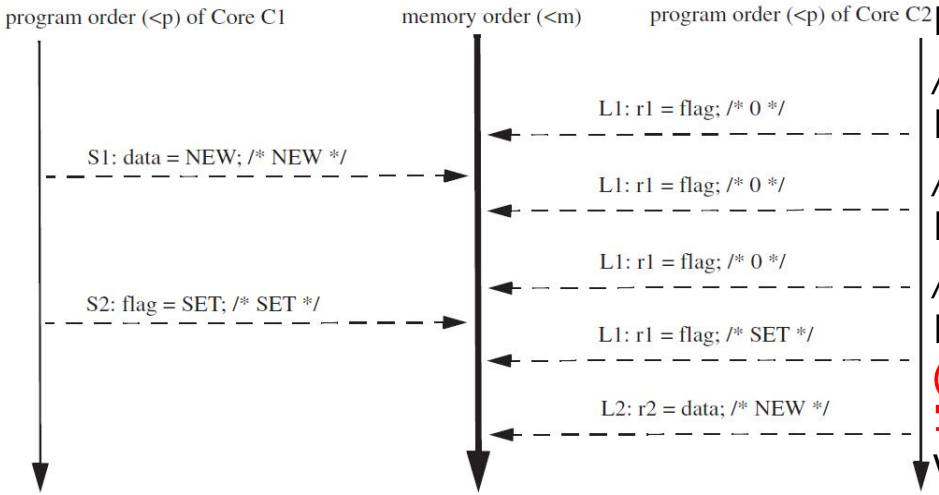


FIGURE 3.1: A Sequentially Consistent Execution of Table 3.1's Program.

(1) 所有core执行的Load/Store满足程序序

/* Load -> Load */

If L(a) <p L(b) => L(a) <m L(b)

/*Load -> Store */

If L(a) <p S(b) => L(a) <m L(b)

/* Store ->Store */

If S(a) <p S(b) => S(a) <m S(b)

/* Store -> Load */

If S(a) <p L(b) => S(a) <m L(b)

(2) 对同一存储单元的Load操作的值来源于最近一次写操作(global memory order)

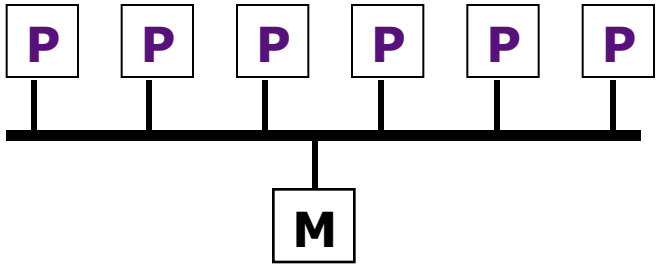
Value of L(a) = Value of $Max_{<m}${S(a) <m L(a)},

$Max_{<m}$表示最近的memory order

Sequential Consistency

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

Issues in Implementing Sequential Consistency



现代计算机系统实现SC 的问题

- *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. Write buffer*

  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 ！ ！ ！

Relaxed Consistency Models

- **Rules:**
  - X ⟶ Y :Operation X must complete before operation Y is done

Sequential consistency requires (SC) :

R ⟶ W, R ⟶ R, W ⟶ R, W ⟶ W

Relax W ⟶ R (TSO)

"Total store ordering" (X86)

Relax W ⟶ W (PSO)

"Partial store order"

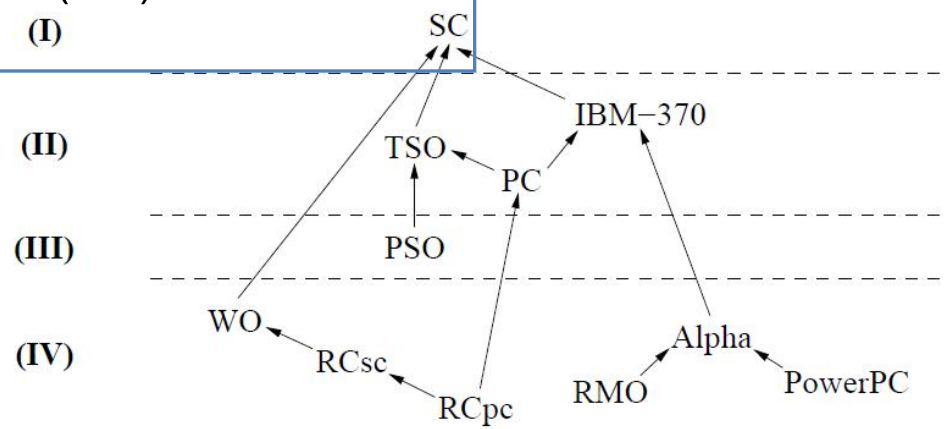


Figure 2.24: Relationship among models according to the "stricter" relation.

Relax R ⟶ W and R ⟶ R

"Weak ordering" and "release consistency"

Relax R ⟶ R , R ⟶ W , W⟶R, W ⟶ W (RMO)

"Release Memory Ordering"

Maintains the program order to access the same location:

W ⟶R, W ⟶ W

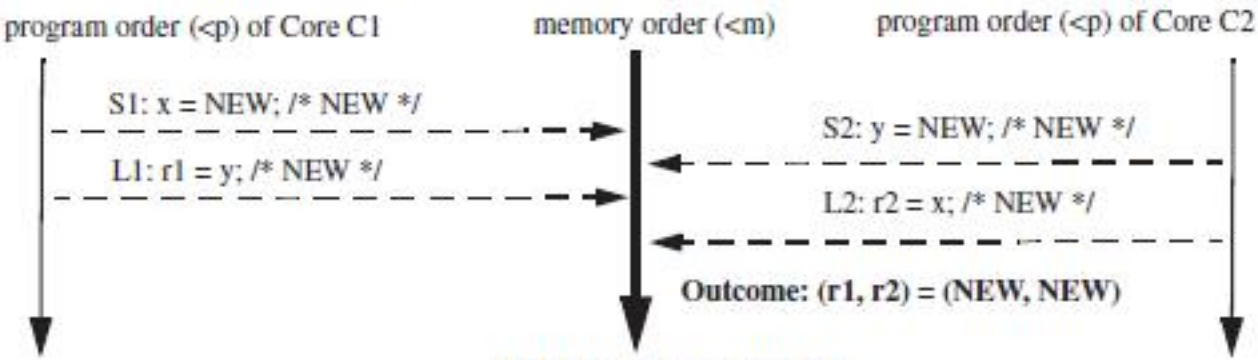
Simple categorization of relaxed models

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

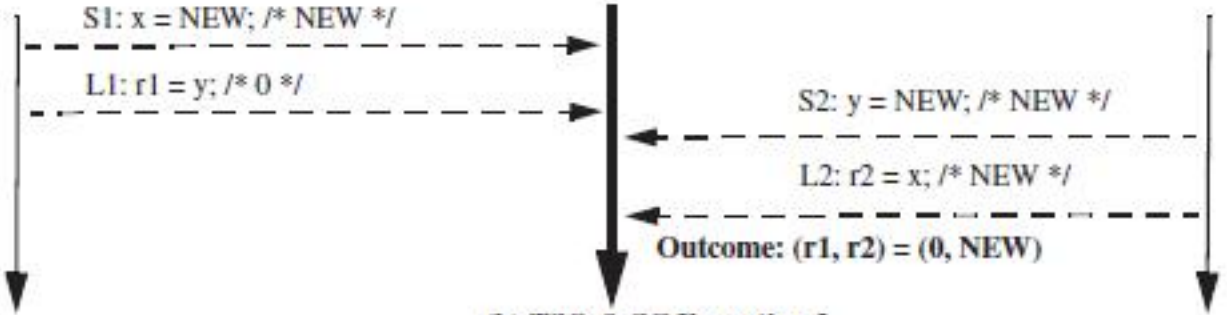
Figure 8: Simple categorization of relaxed models. A √ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The "Read Own Write Early" relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The "Read Others' Write Early" relaxation is possible and detectable with complex implementations of RCsc.

TABLE 4.1: Can Both r1 and r2 be Set to 0?

Core C1	Core C2	Comments
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0*/



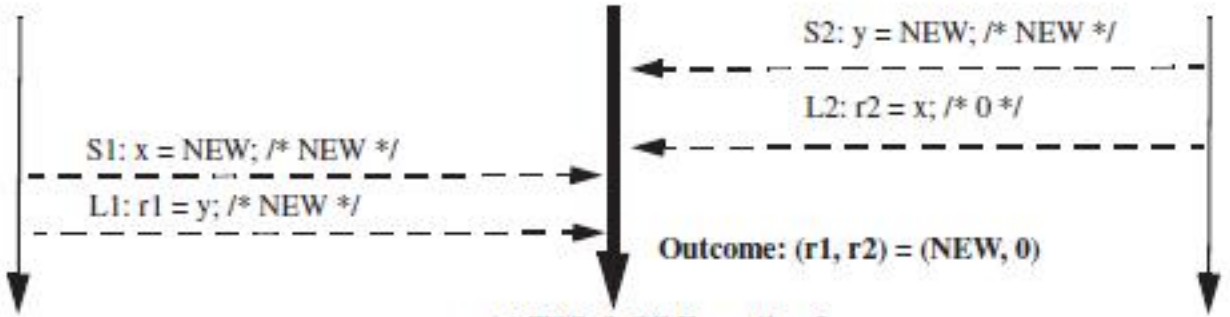
(a) TSO & SC Execution 1



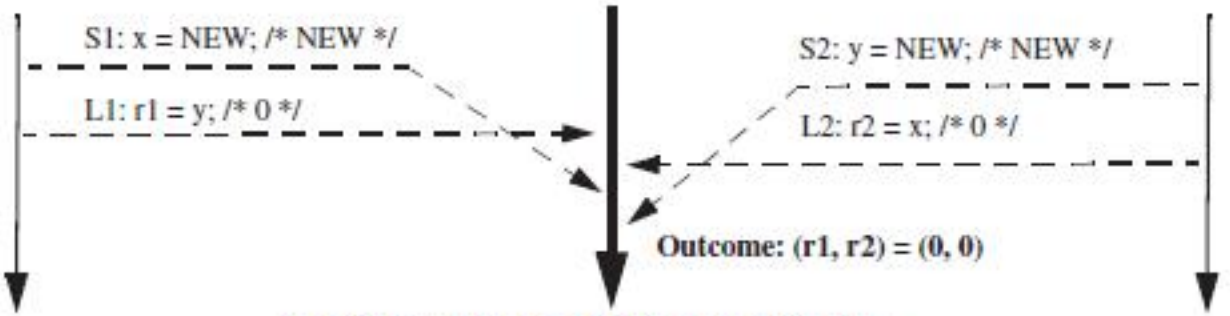
(b) TSO & SC Execution 2

TABLE 4.1: Can Both r1 and r2 be Set to 0?

Core C1	Core C2	Comments
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0*/



(c) TSO & SC Execution 3



(d) TSO Execution, but NOT an SC Execution

RE 4.2: Four Alternative TSO Executions of Table 4.1's Program.

TABLE 4.3: Can r1 or r3 be Set to 0?		
Core C1	**Core C2**	**Comments**
S1: x = NEW; L1: r1 = x; L2: r2 = y;	S2: y = NEW; L3: r3 = y; L4: r4 = x;	/* Initially, x = 0 & y = 0*/ /* Assume r2 = 0 & r4 = 0 */

- **当(r2, r4) = (0, 0) 时， (r1, r3) 一定为 (0, 0) ？**

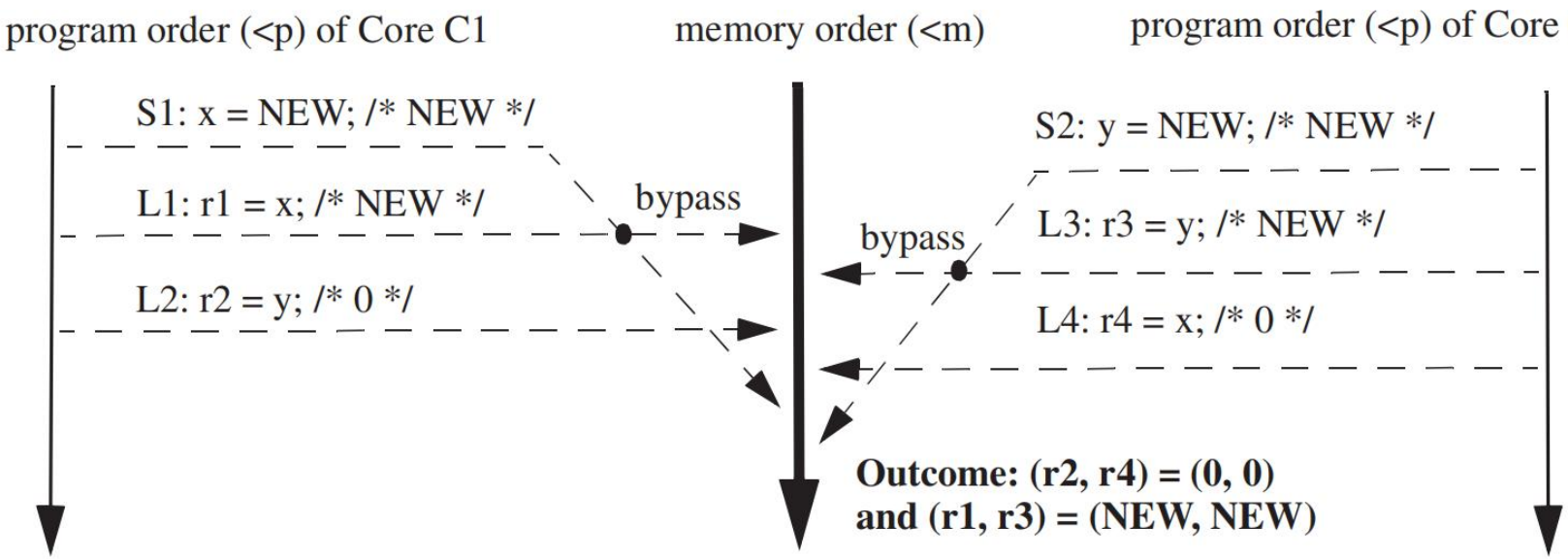


FIGURE 4.3: A TSO Execution of Table 4-3's Program (with "bypassing").

Memory Fences

Instructions to sequentialize memory accesses

实现弱一致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

Table 1: Summary of Memory Ordering

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

7.5 同步问题

01 **生产者-消费者问题**

02 互斥问题

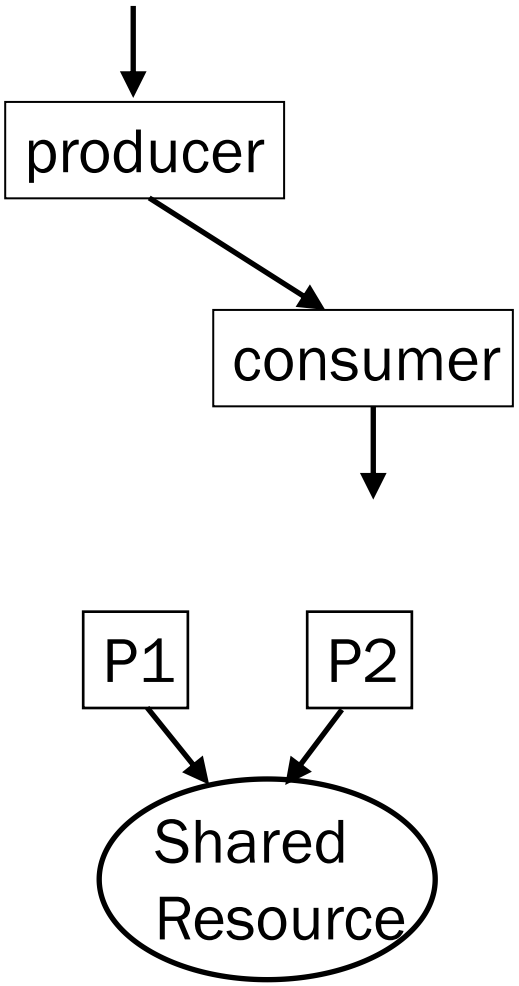
Synchronization

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

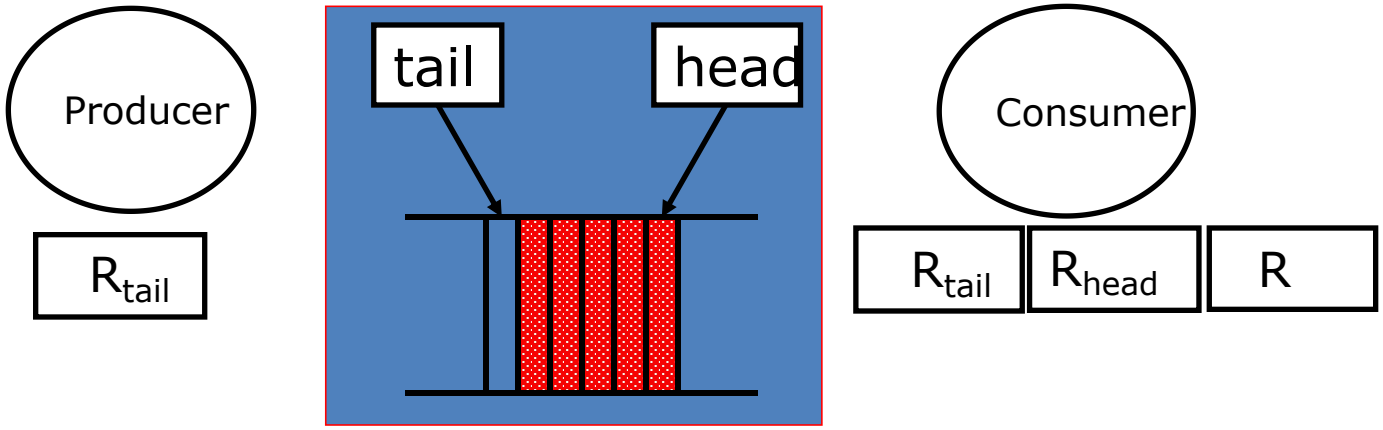
总体上存在两类同步操作问题:

*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?

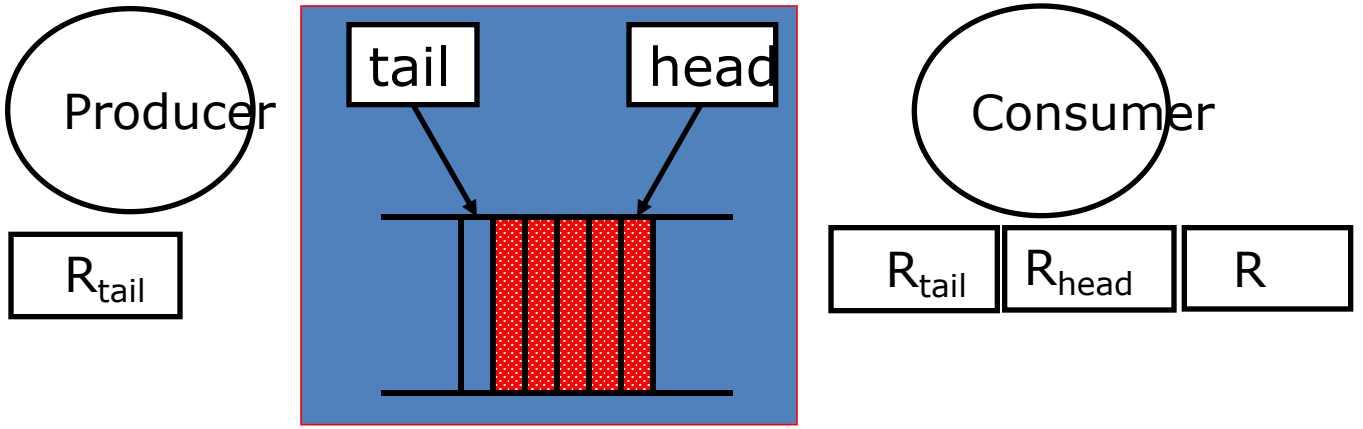
Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:

2, 3, 4, 1

4, 1, 2, 3

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
```

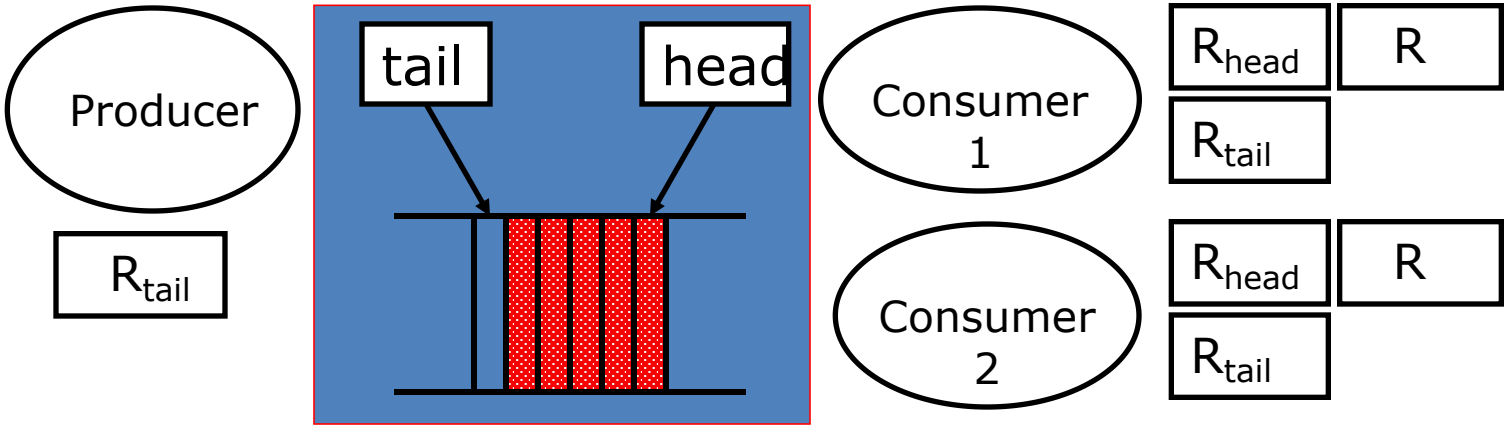
ensures that tail ptr is not updated before x has been stored

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:

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)

Critical section:
Needs to be executed atomically by one consumer

What is wrong with this code?

7.5 同步问题

01 生产者-消费者问题

02 **互斥问题**

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

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
```

- 为避免死锁，我们让一进程等待时放弃reservation（预订）
  - Process 1 sets c1 to 0.
- 死锁显然是没有了，但有可能会发生 活锁*(livelock)* 现象
  - C1 = 1, C2=1, Read C2, Read C1, C1 =0, C2 = 0, C1=1, C2=1, C1=0, C2=0
- 可能还会出现某个进程始终无法进入临界区 (*starvation)*
  - *例如： C1=1，C2= 1，C1=0，(P2)Read C1 进入临界区， P1和P2竞争 P2始终胜出*

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 和 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)和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

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，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;
```

Implementation of Semaphores

在顺序一致性模型中，信号量 (mutual exclusion) 可以用常规的 Load 和 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), Rv, R:
    R ← M[m];
    M[m] ← R + Rv;
```

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

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)和 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:    Load Rhead, (head)
spin:   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)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**