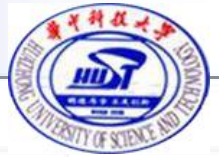
第8章 多处理器

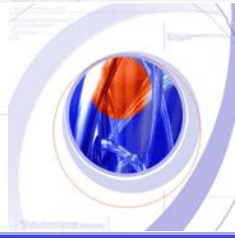


8.1 多处理器概念

8.2 一致性问题

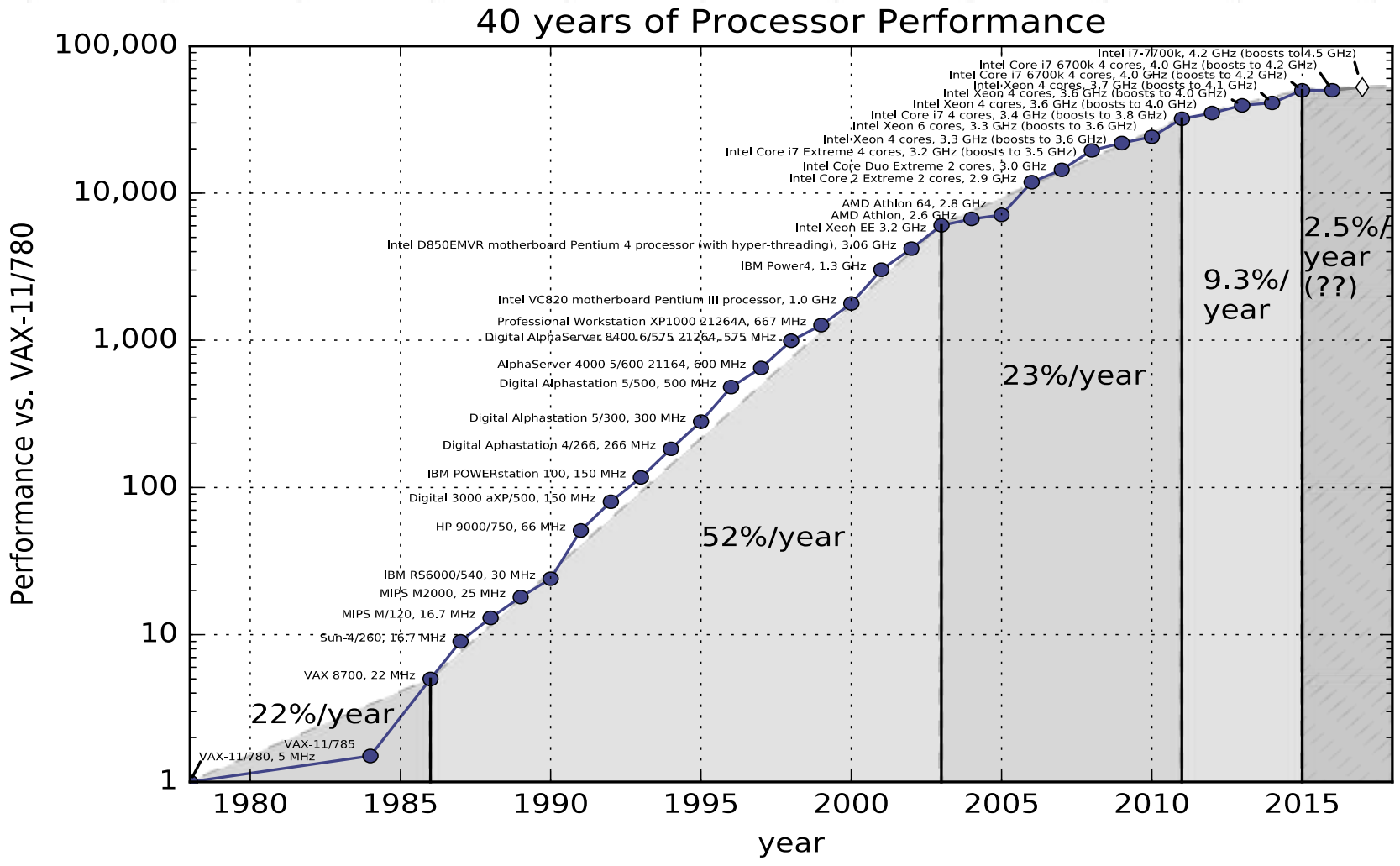
8.3 同步

8.4 同步性能问题



8.1.1多处理器背景

1. 单处理机系统结构性能提升的边际效应不断降低



8.1.1 多处理器背景

2.通过增加处理器核数提升整体计算能力

（1）摩尔定律推动晶体管数量每2年增加2倍左右

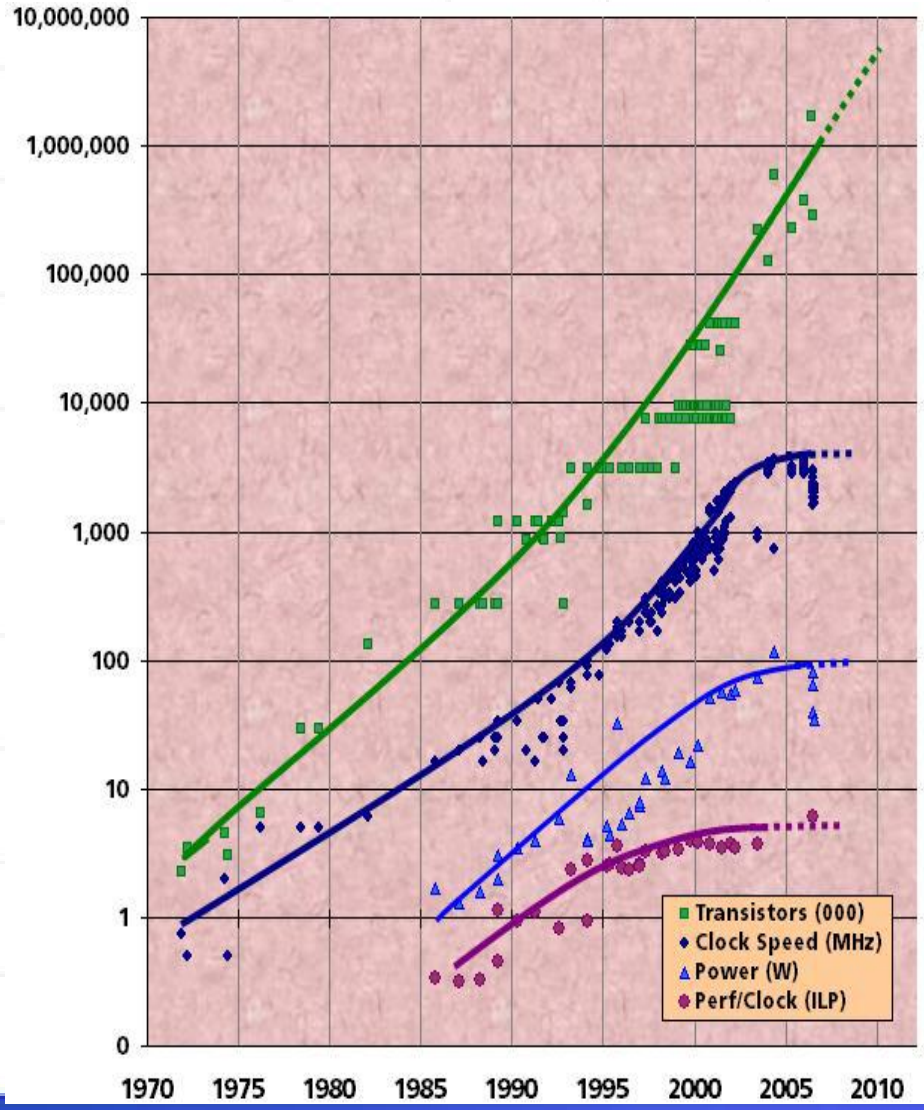
（2）时钟频率几乎不变

（3）处理器功率几乎不变

（4）指令并行度几乎不变

本章重点：中小规模多处理器（处理器或核数＜32）

（多处理机设计的主流）



8.1.1多处理器背景

3. 并行计算机系统结构的分类

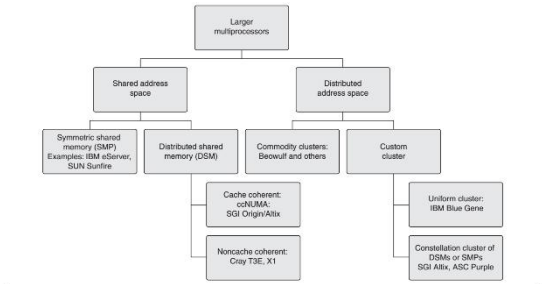


Figure I.21 The space of large-scale multiprocessors and the relation of different classes.

1. Flynn分类法

SISD、SIMD、MISD、MIMD

2. MIMD已成为通用多处理机系统结构的选择，原因：

- MIMD具有灵活性；
- 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。

计算机机群系统（cluster）是一类广泛被采用的MIMD机器。

8.1.2多处理器概念

3. 根据存储器的组织结构，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）

- 集中式共享存储器结构
  - 各处理器共享一个集中式的物理存储器最多由几十个处理器构成
  - 各处理器共享一个集中式的物理存储器。

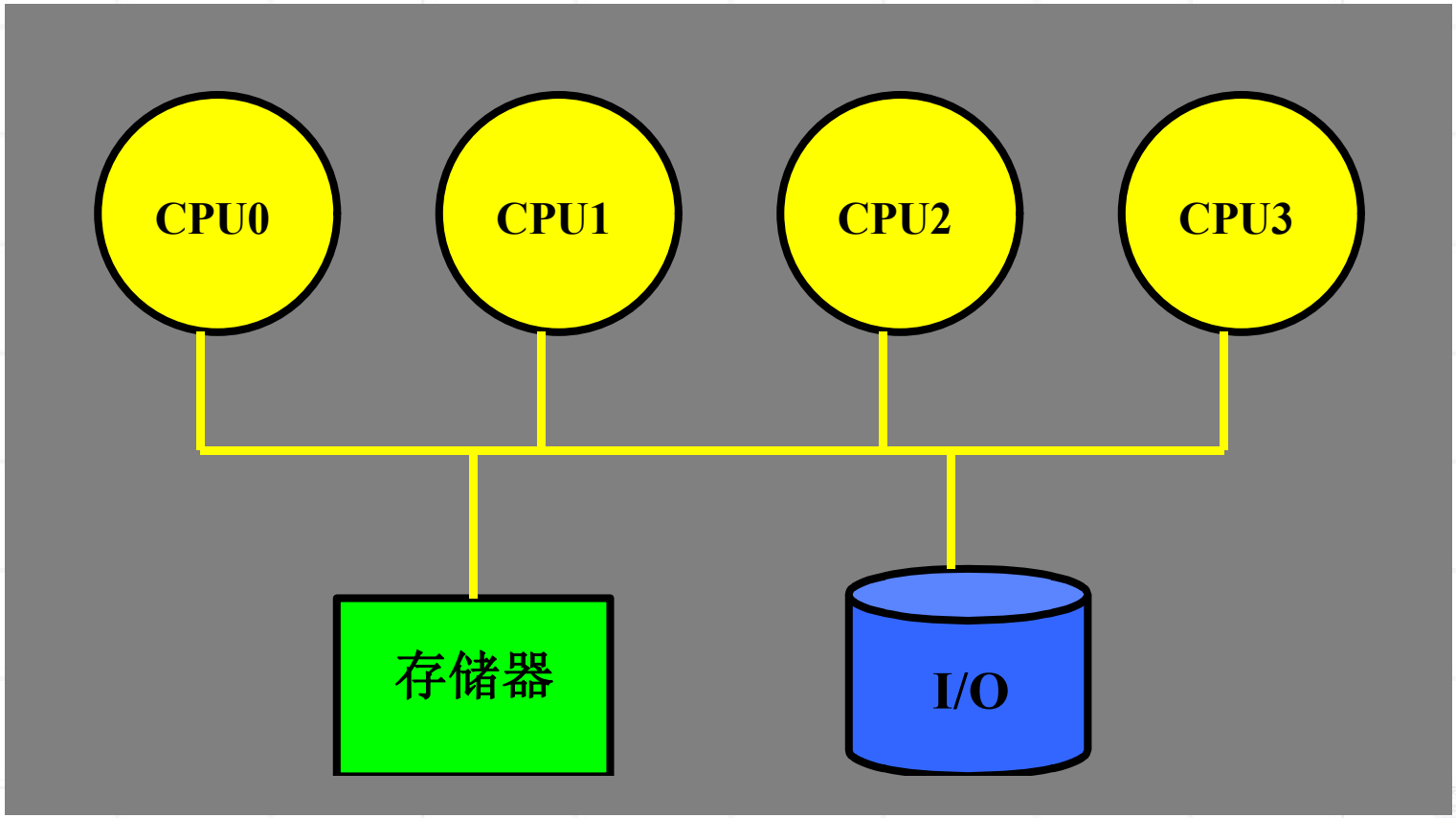
这类机器有时被称为

- SMP机器

（Symmetric shared-memory MultiProcessor）

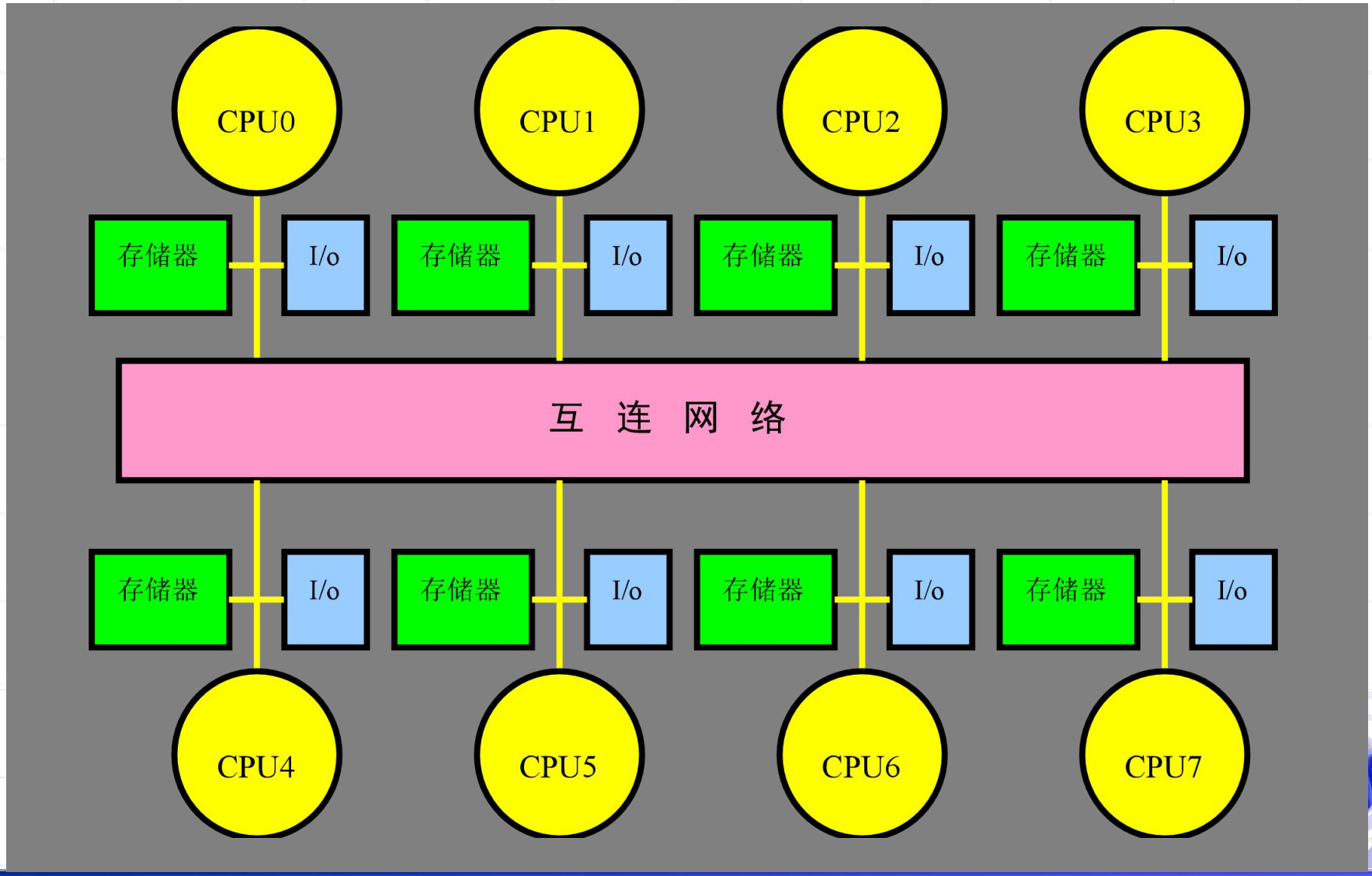
- UMA机器（Uniform Memory Access）在UMA中，所有的处理器平等的访问内存，因此CPU对内存的存取在速度上等是没有差异的，这也是称为统一内存访问的原因。各处理器与内存单元通过互联总线进行连接，各个CPU之间没有主从关系。

8.1.2多处理器概念



对称式共享存储器多处理机的基本结构

8.1.2多处理器概念


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

8.1.2多处理器概念

- 将存储器分布到各结点有两个优点
  - 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；
  - 对本地存储器的访问延迟时间小。
- 最主要的缺点
  - 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。

8.1.2多处理器概念

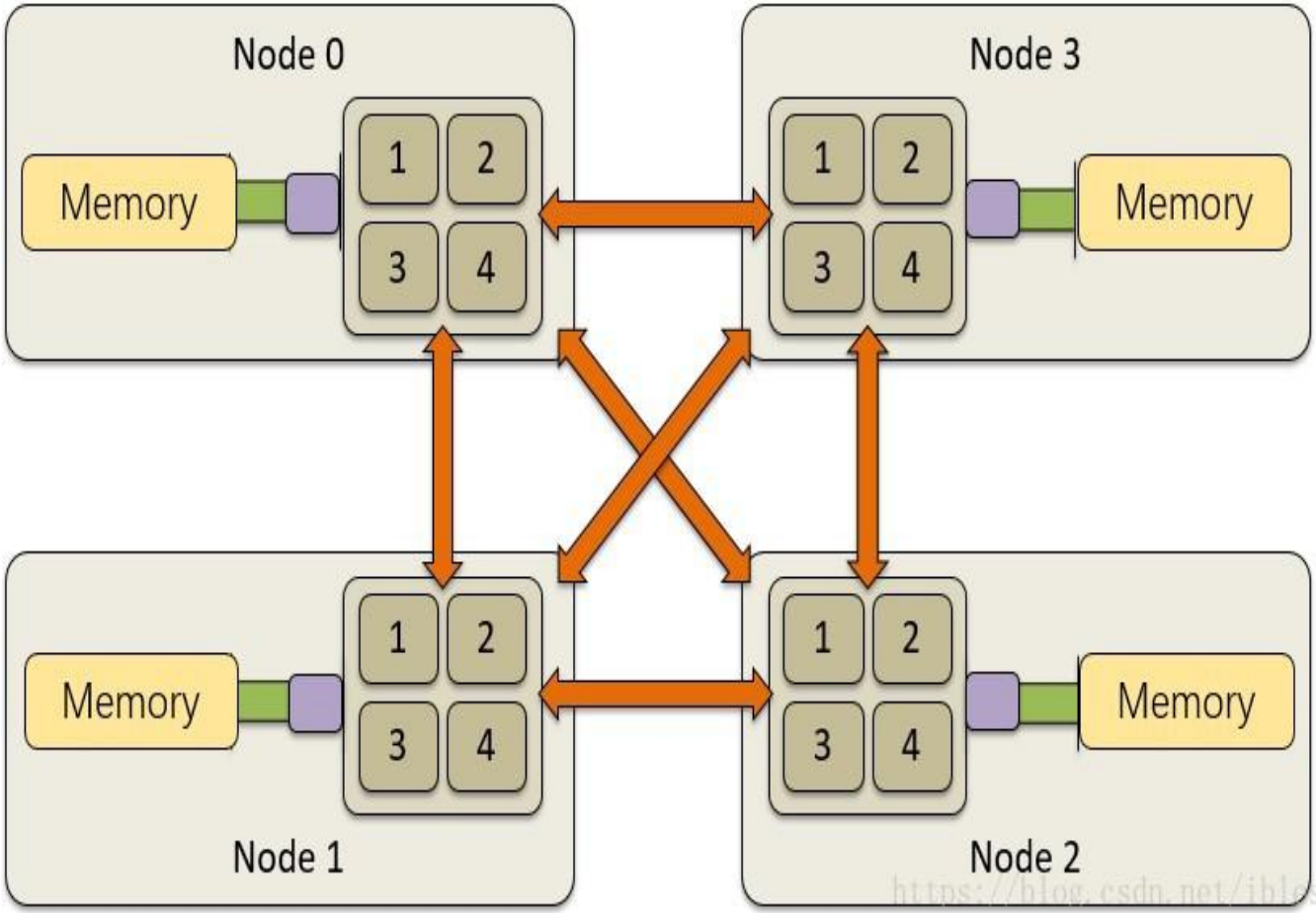
存储器系统结构和通信机制

1. 两种存储器系统结构和通信机制

- 共享全局地址空间
  - 物理共享的存储器具有统一逻辑地址
  - 物理上分离的所有存储器作为一个统一的共享逻辑空间进行编址。分布式共享存储器系统

    (DSM: Distributed Shared-Memory)
- 分布式的独立地址空间
  - 整个系统的地址空间由多个独立逻辑地址空间构成
  - 不同结点中的地址空间之间是相互独立的
  - 每个结点中的存储器只能由本地的处理器进行访问，远程的处理器不能直接对其进行访问。

8.1.2多处理器概念



4路多核处理服务器结构

NUMA机器 （NUMA: Non-Uniform Memory Access）

8.1.2多处理器概念

2. 通信机制

- 共享存储器通信机制
  - 共享地址空间的计算机系统采用
  - 处理器之间是通过用load和store指令对相同存储器地址进行读/写操作来实现的。
- 消息传递通信机制
  - 多个独立地址空间的计算机采用
  - 通过处理器间显式地传递消息来完成
  - 消息传递多处理机中，处理器之间是通过发送消息来进行通信的，这些消息请求进行某些操作或者传送数据。

8.1.2多处理器概念

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：

- 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)

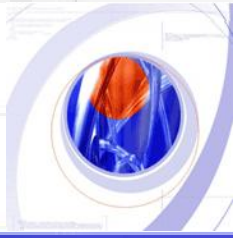
- 目的处理器接收到消息以后，执行相应的操作或代替远程处理器进行访问，并发送一个应答消息将结果返回。

- 同步消息传递

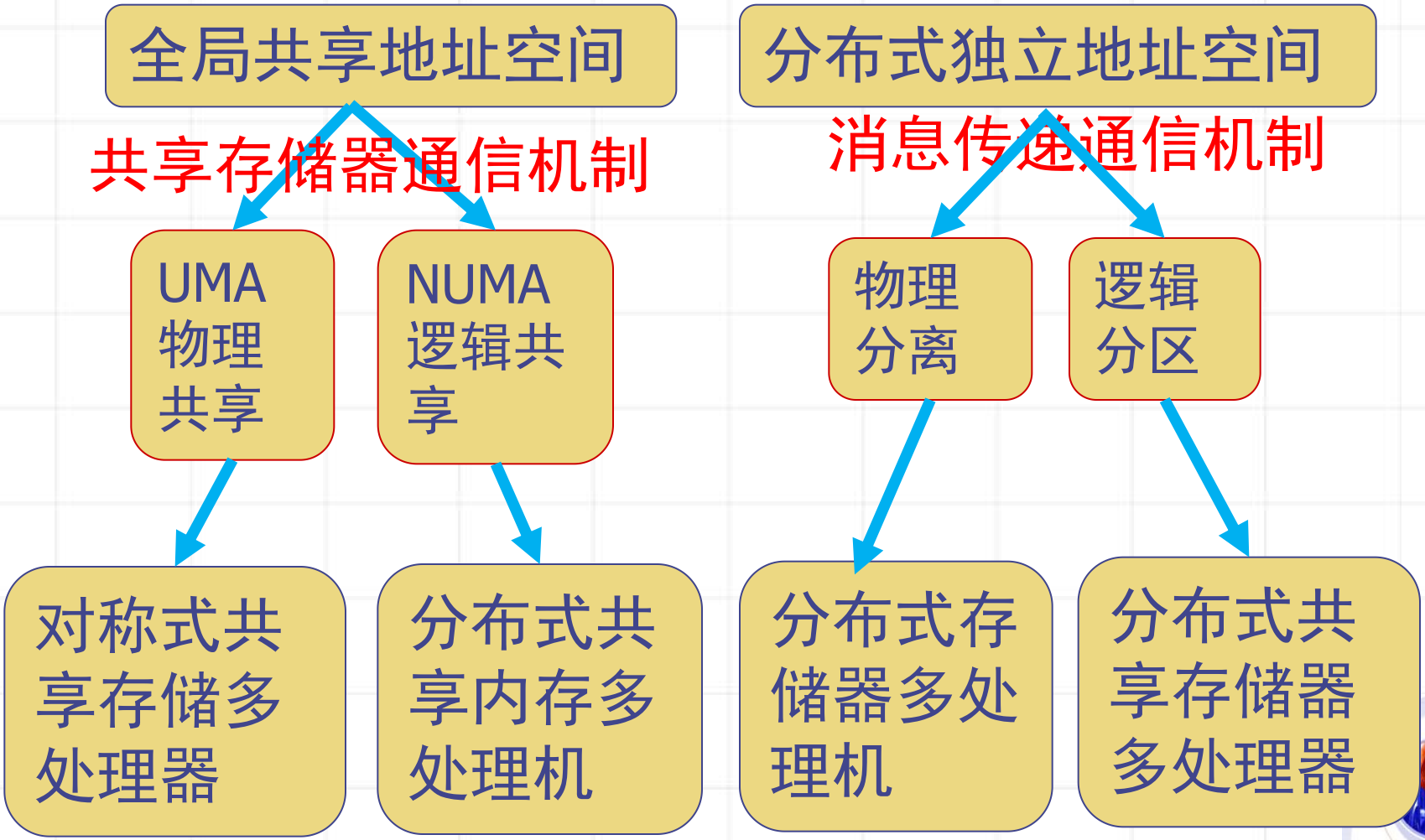
请求处理器发送一个消息后一直要等到应答结果才继续运行。

- 异步消息传递

当请求处理器发送一个消息后可以处理其他事情，数据发送方在得到所需数据后，通知请求处理器。



8.1.2多处理器概念

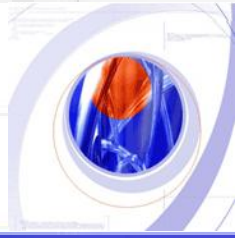

全局共享地址空间
分布式独立地址空间
共享存储器通信机制
消息传递通信机制
UMA
物理
共享
NUMA
逻辑共
享
物理
分离
逻辑
分区
对称式共
享存储多
处理器
分布式共
享内存多
处理机
分布式存
储器多处
理机
分布式共
享存储器
多处理器

3. 不同通信机制的优点

- 共享存储器通信的主要优点
  - 与常用的对称式多处理机使用的通信机制兼容。
  - 易于编程，同时在简化编译器设计方面也占有优势。
  - 采用大家所熟悉的共享存储器模型开发应用程序，而把重点放到解决对性能影响较大的数据访问上。
  - 当通信数据量较小时，通信开销较低，带宽利用较好。
  - 可以通过采用Cache技术来减少远程通信的频度，减少了通信延迟以及对共享数据的访问冲突。

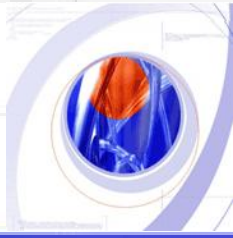
8.1.2多处理器概念

- 消息传递通信机制的主要优点
  - 硬件较简单。
  - 通信是显式的，因此更容易搞清楚何时发生通信以及通信开销是多少。
  - 显式通信可以让编程者重点注意并行计算的主要通信开销，使之有可能开发出结构更好、性能更高的并行程序。
  - 同步很自然地与发送消息相关联，能减少不当的同步带来错误的可能性。



8.1.2多处理器概念

- 可在支持上面任何一种通信机制的硬件模型上建立所需的通信模式平台。
  - 在共享存储器上支持消息传递相对简单。
  - 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。



8.1.3 多处理器整体挑战

并行处理面临的挑战

并行处理面临着两个重要的挑战

- 程序中的并行性有限
- 相对较大的通信开销

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

8.1.3 多处理器整体挑战

1. 第一个挑战

有限的并行性使计算机要达到很高的加速比十分困难。

例8.1 假设想用100个处理器达到80的加速比，求原计算程序中串行部分最多可占多大的比例？

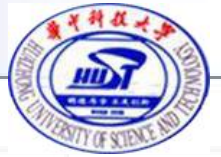
解 Amdahl定律为：

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

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

由上式可得：并行比例＝0.9975

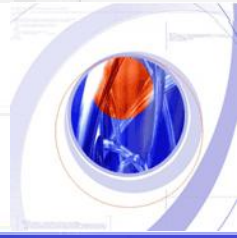
8.1.3 多处理器整体挑战



2. 第二个挑战：多处理机中远程访问的延迟较大

- 在现有的机器中，处理器之间的数据通信大约需要50～1000个时钟周期。
- 主要取决于：

  通信机制、互连网络的种类和机器的规模
- 在几种不同的共享存储器并行计算机中远程访问一个字的典型延迟



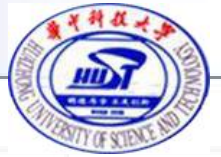
8.1.3 多处理器整体挑战

机器	通信机制	互连网络	处理机最大数量	典型远程存储器访问时间（ns）
Sun Starfire servers	SMP	多总线	64	500
SGI Origin 3000	NUMA	胖超立方体	512	500
Cray T3E	NUMA	3维环网	2048	300
HP V series	SMP	8×8交叉开关	32	1000
HP AlphaServer GS	SMP	开关总线	32	400

8.1.3 多处理器整体挑战

例8.2 假设有一台32台处理器的多处理机，对远程存储器访问时间为200ns。除了通信以外，假设所有其它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。处理器的时钟频率为2GHz，如果指令基本的CPI为0.5（设所有访存均命中Cache），求在没有远程访问的情况下和有0.2%的指令需要远程访问的情况下，前者比后者快多少?

8.1.3 多处理器整体挑战



解 有0.2%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

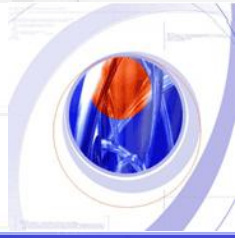
＝0.5＋0.2%×远程访问开销

远程访问开销为：

远程访问时间/时钟周期时间＝200ns/0.5ns＝400个时钟周期

∴ CPI＝0.5＋0.2%×400＝1.3

因此在没有远程访问的情况下的机器速度是有0.2%远程访问的机器速度的1.3/0.5=2.6倍。



8.1.3 多处理器整体挑战

- 问题的解决
  - 并行性不足： 采用并行性更好的算法
  - 远程访问延迟的降低：靠系统结构支持和编程技术

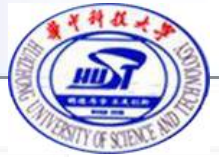
3. 在并行处理中，影响性能（负载平衡、同步和存储器访问延迟等）的关键因素常依赖于：

应用程序的高层特性

如数据的分配，并行算法的结构以及在空间和时间上对数据的访问模式等。

- 依据应用特点可把多机工作负载大致分成两类：
  - 单个程序在多处理机上的并行工作负载
  - 多个程序在多处理机上的并行工作负载

第8章 多处理器

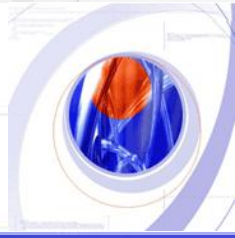


8.1 多处理器概念

8.2 一致性问题

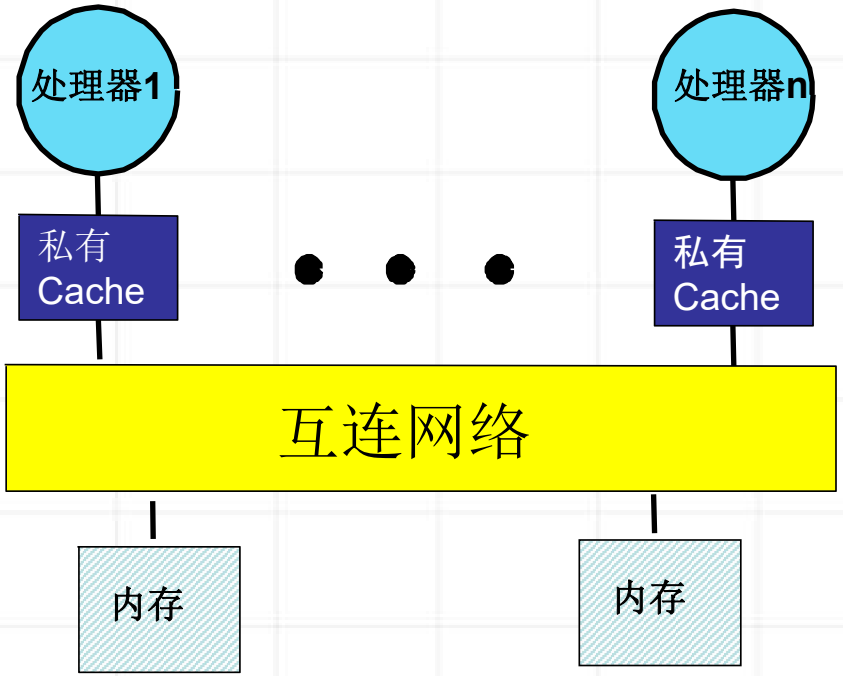
8.3 同步

8.4 同步性能问题

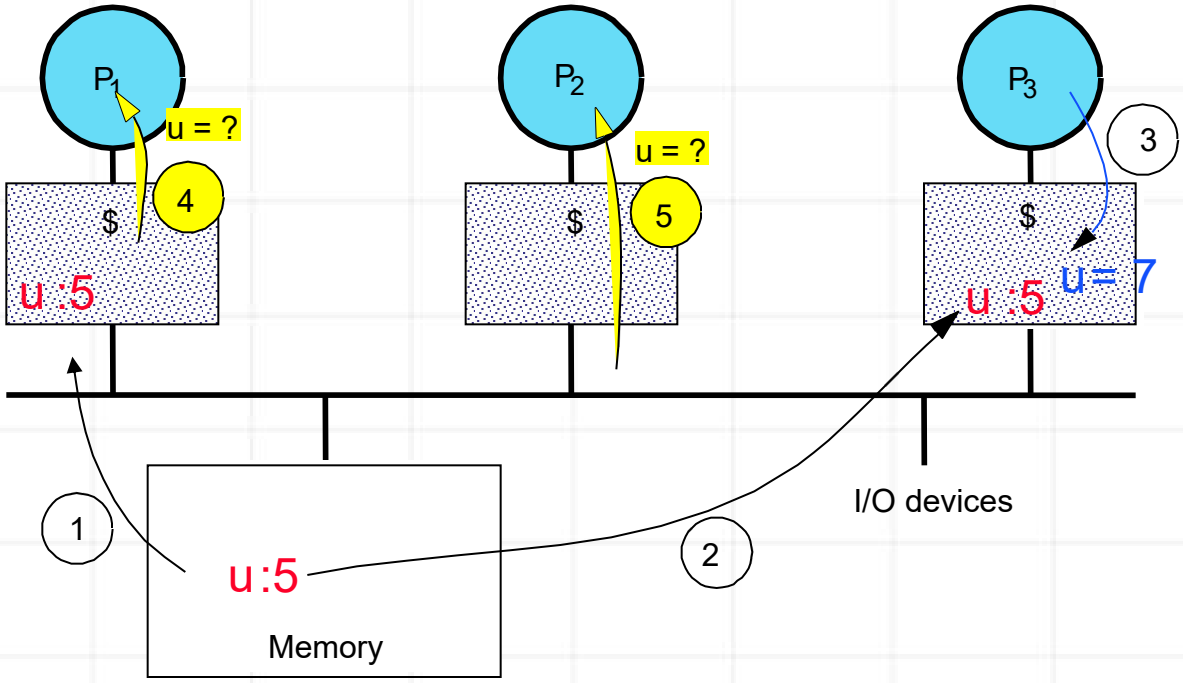


多处理机Cache一致性

对称式共享存储器系统结构中，多个处理器共享一个存储器。



8.2.1 多处理器Cache一致性



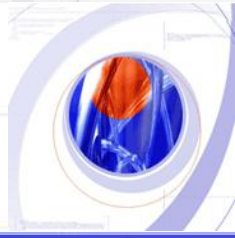
- 允许共享数据进入Cache，就可能出现多个处理器的Cache中都有同一存储块的副本，
- 当其中某个处理器对其Cache中的数据进行修改后，就会使得其Cache中的数据与其他Cache中的数据不一致

8.2.1 多处理器Cache一致性

2. 存储器的一致性

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
  - What: 读操作得到的是什么值
  - When: 什么时候才能将已写入的值返回给读操作
- 需要满足以下条件
  - 处理器P对单元X进行一次写之后又对单元X进行读，读和写之间没有其它处理器对单元X进行写，则P读到的值总是前面写进去的值。



8.2.1 多处理器Cache一致性

- 处理器P对单元X进行写之后，另一处理器Q对单元X进行读，读和写之间无其它写，则Q读到的值应为P写进去的值。
- 对同一单元的写是串行化的，即任意两个处理器对同一单元的两次写，从各个处理器的角度看来顺序都是相同的。（写串行化 ）

➢ 在后面的讨论中，我们假设：

- 直到所有的处理器均看到了写的结果，这个写操作才算完成；
- 处理器的任何访存均不能改变写的顺序。就是说，允许处理器对读进行重排序，但必须以程序规定的顺序进行写。

8.2.2 一致性监听协议概念

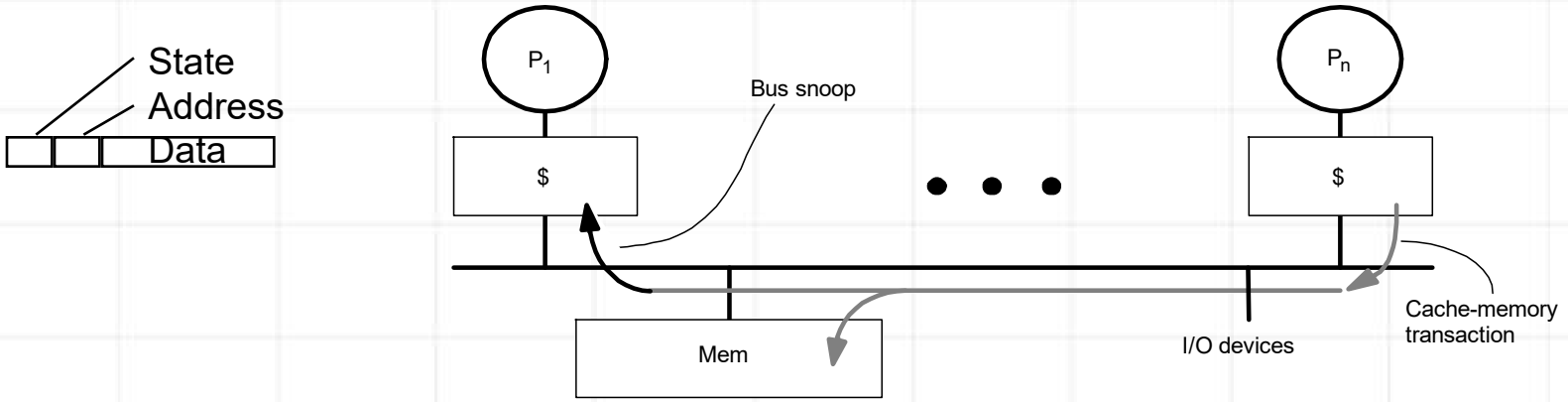
1. Cache一致性协议

在多个处理器中用来维护一致性的协议。

- 关键：跟踪记录共享数据块的状态
- 两类协议（采用不同的技术跟踪共享数据的状态）
  - 监听式协议（snooping）
  - 目录式协议（directory）

8.2.2 一致性监听协议概念

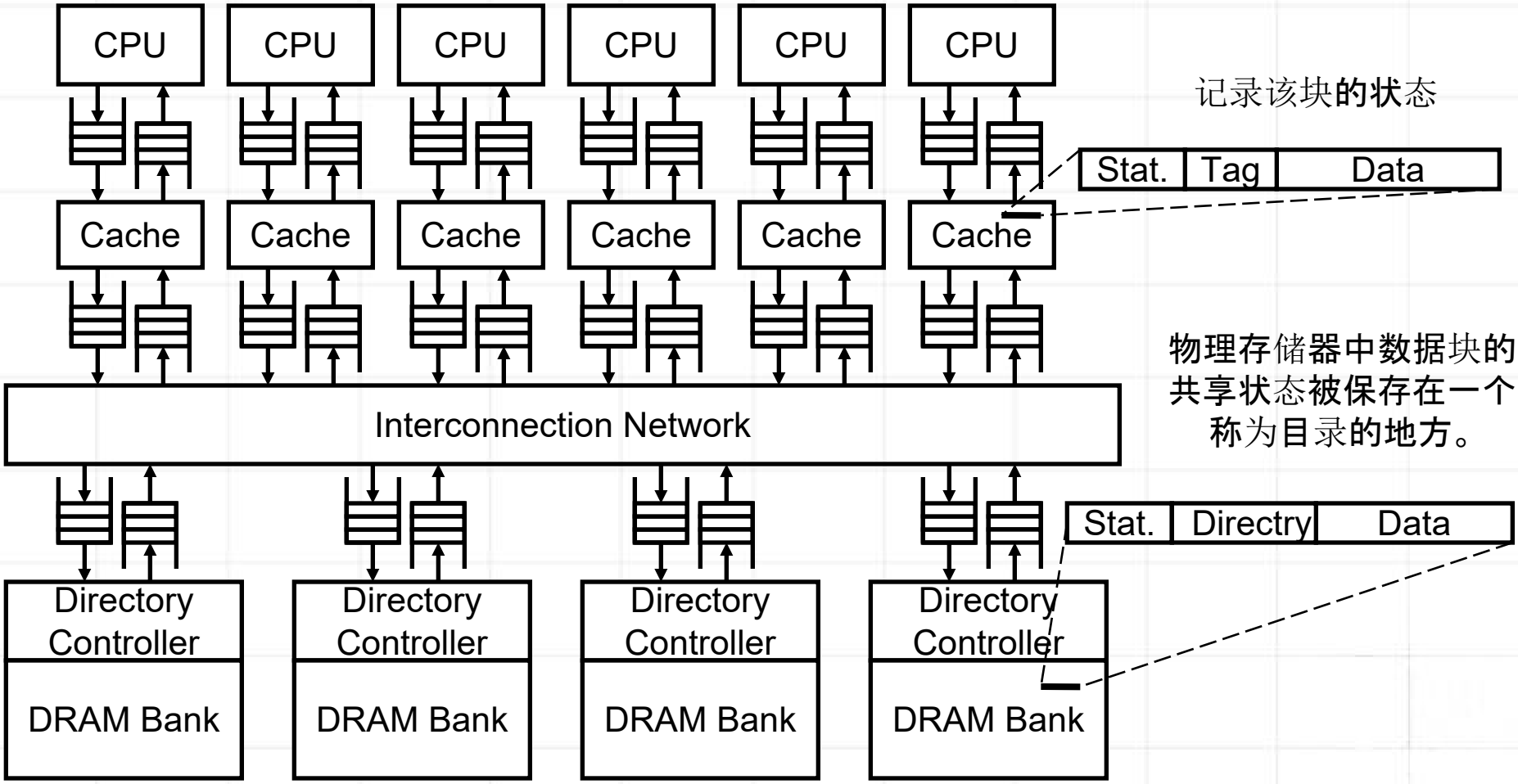
监听式协议（snooping）



- **Cache**通常连在共享存储器的总线上，当某个**Cache**需要访问存储器时，它会把请求放到总线上广播出去，其他各个**Cache**控制器通过监听总线（它们一直在监听）来判断它们是否有总线上请求的数据块。如果有，就进行相应的操作。
- 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。
-

8.2.2 一致性监听协议概念

目录式协议(Directory)



8.2.2 一致性监听协议概念

多处理器写更新操作处理。

- 写作废协议

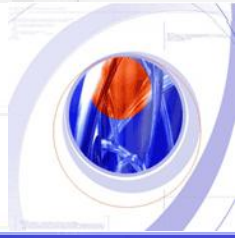
  在处理器对某个数据项进行写入之前，保证它拥有对该数据项的唯一的访问权。(作废其它的副本)□

- 写更新协议

  当一个处理器对某数据项进行写入时，通过广播使其它Cache中所有对应于该数据项的副本进行更新。

存储器写策略

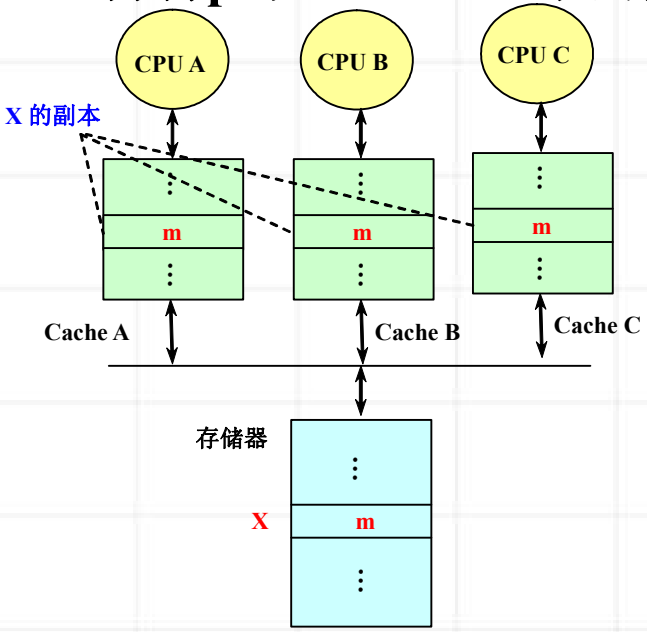
写直达操作和写回操作



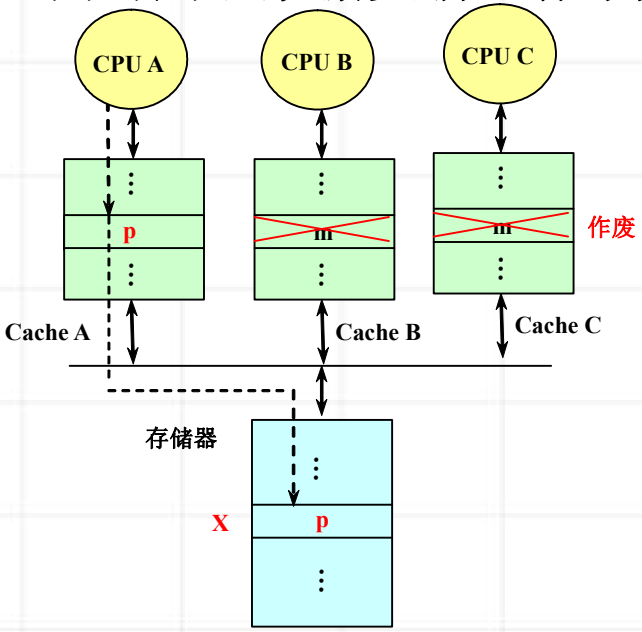
8.2.2 一致性监听协议概念

例 监听总线、写作废协议举例（采用写直达法）

初始状态：CPU A、CPU B、CPU C都有X的副本。在CPU A要对X进行写入时，需先作废CPU B和CPU C中的副本，然后再将p写入Cache A中的副本，同时用该数据更新主存单元X。



（a） CPU A 写入前



（b） CPU A 将 p 写入 X 后，作废其他 Cache 中的副本

8.2.2 一致性监听协议概念

- 写更新协议

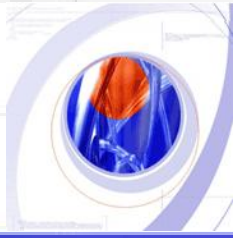
当一个处理器对某数据项进行写入时，通过广播使其它Cache中所有对应于该数据项的副本进行更新。

例 监听总线、写更新协议举例（采用写直达法）

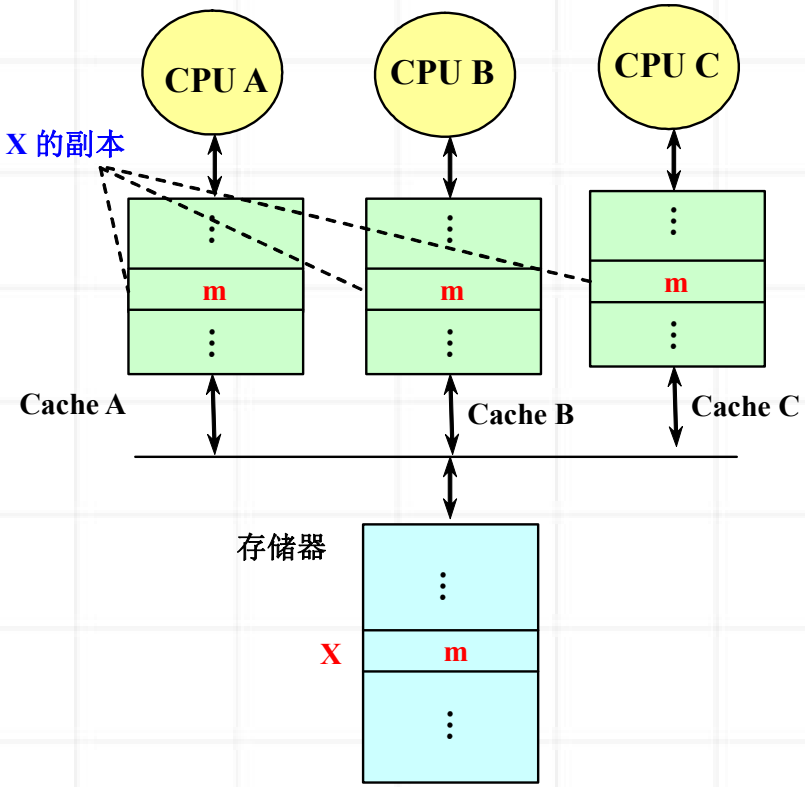
假设：3个Cache都有X的副本。

当CPU A将数据p写入Cache A中的副本时，将p广播给所有的Cache，这些Cache用p更新其中的副本。

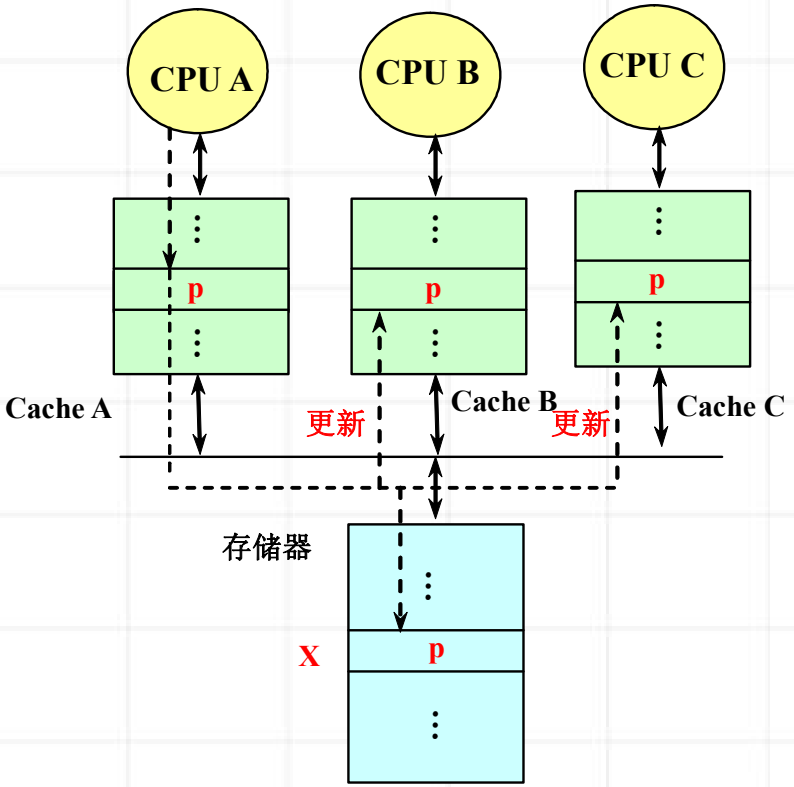
由于这里是采用写直达法，所以CPU A还要将p写入存储器中的X。如果采用写回法，则不需要写入存储器。



8.2.2 一致性监听协议概念



（a） CPU A 写入前



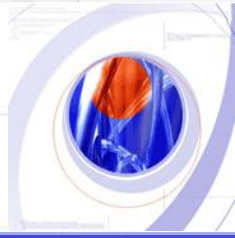
（b） CPU A 将 p 写入 X 后，更新其他 Cache 中的副本

8.2.2 一致性监听协议概念

- 写更新和写作废协议性能上的差别主要来自：
  - 在对同一个数据进行多次写操作而中间无读操作的情况下，写更新协议需进行多次写广播操作，而写作废协议只需一次作废操作。
  - 在对同一Cache块的多个字进行写操作的情况下，写更新协议对于每一个写操作都要进行一次广播，而写作废协议仅在对该块的第一次写时进行作废操作即可。

    写作废是针对Cache块进行操作，而写更新则是针对字（或字节）进行。
  - 考虑从一个处理器A进行写操作后到另一个处理器B能读到该写入数据之间的延迟时间。

    写更新协议的延迟时间较小。



8.2.3 写直达作废一致性协议

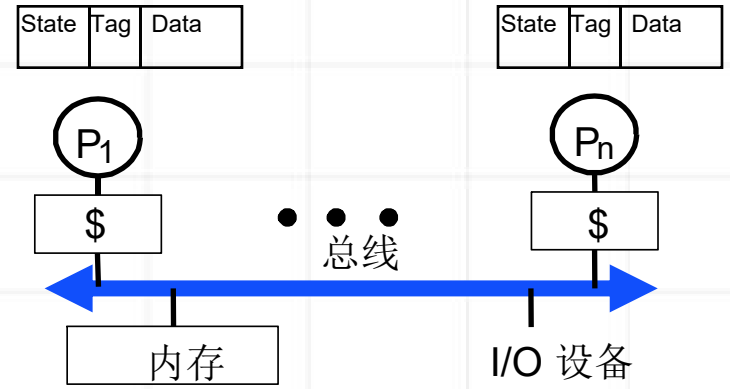
监听协议的实现

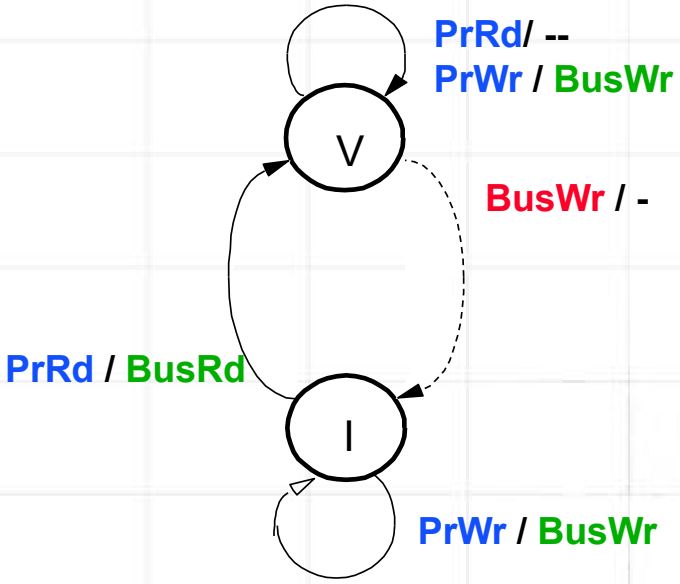
- 在每个结点内嵌入一个有限状态控制器。
  - 该控制器根据来自处理器或总线的请求以及Cache块的状态，做出相应的响应。
- 实现监听协议的关键有3个方面
  - 处理器之间通过一个可以实现广播的互连机制相连。

    通常采用的是总线。
  - 当一个处理器的Cache响应本地CPU的访问时，如果它涉及到全局操作，其Cache控制器就要在获得总线的控制权后，在总线上发出相应的消息。
  - 所有处理器都一直在监听总线，它们检测总线上的地址在它们的Cache中是否有副本。若有，则响应该消息，并进行相应的操作 。
- 写操作的串行化：由总线实现　　获取总线控制权的顺序性

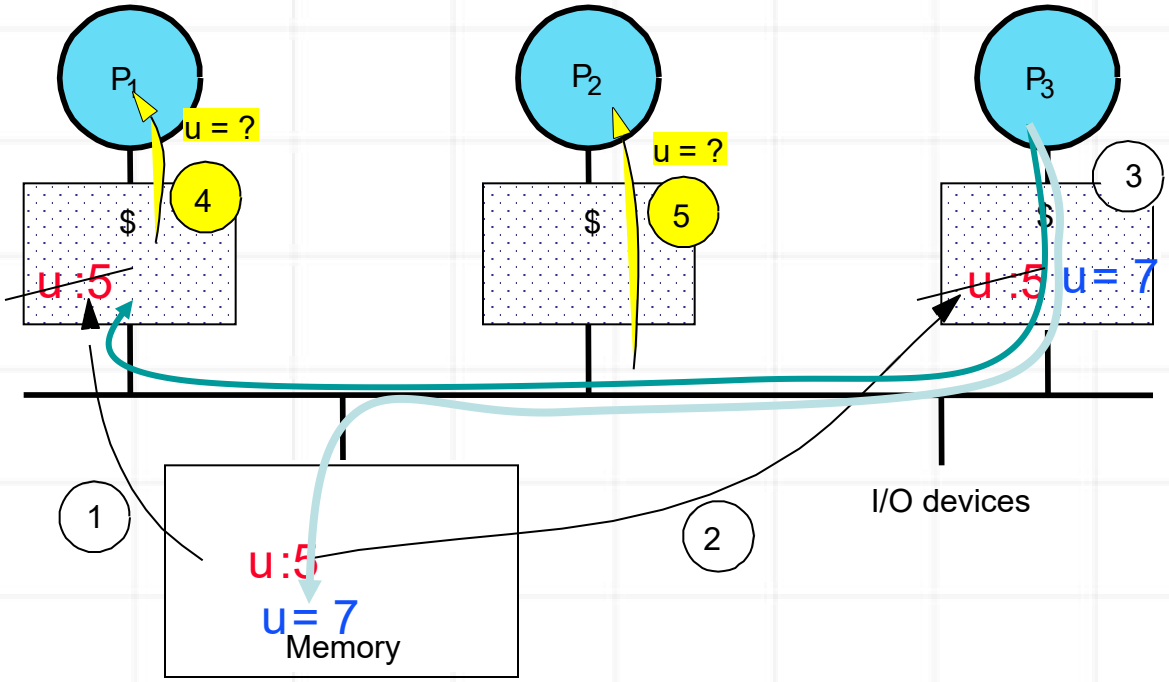
8.2.3 写直达作废一致性协议

- 基本总线协议
  - 每个处理器都有单独的cache和状态标记
  - 经过总线的所有事务都被监听
- 写操作使所有其他缓存失效
  - 可以同时存在对数据块的多个读操作，但写操作会使所有对该块的缓存全部失效
- 每个缓存中的每个数据块都存在两种状态
  - 对于单一处理器而言
  - 数据块的状态由一个p向量表示
  - 硬件状态位与缓存中的数据块相关联
  - 其他数据块在该缓存中可以被视为无效(不存在)状态



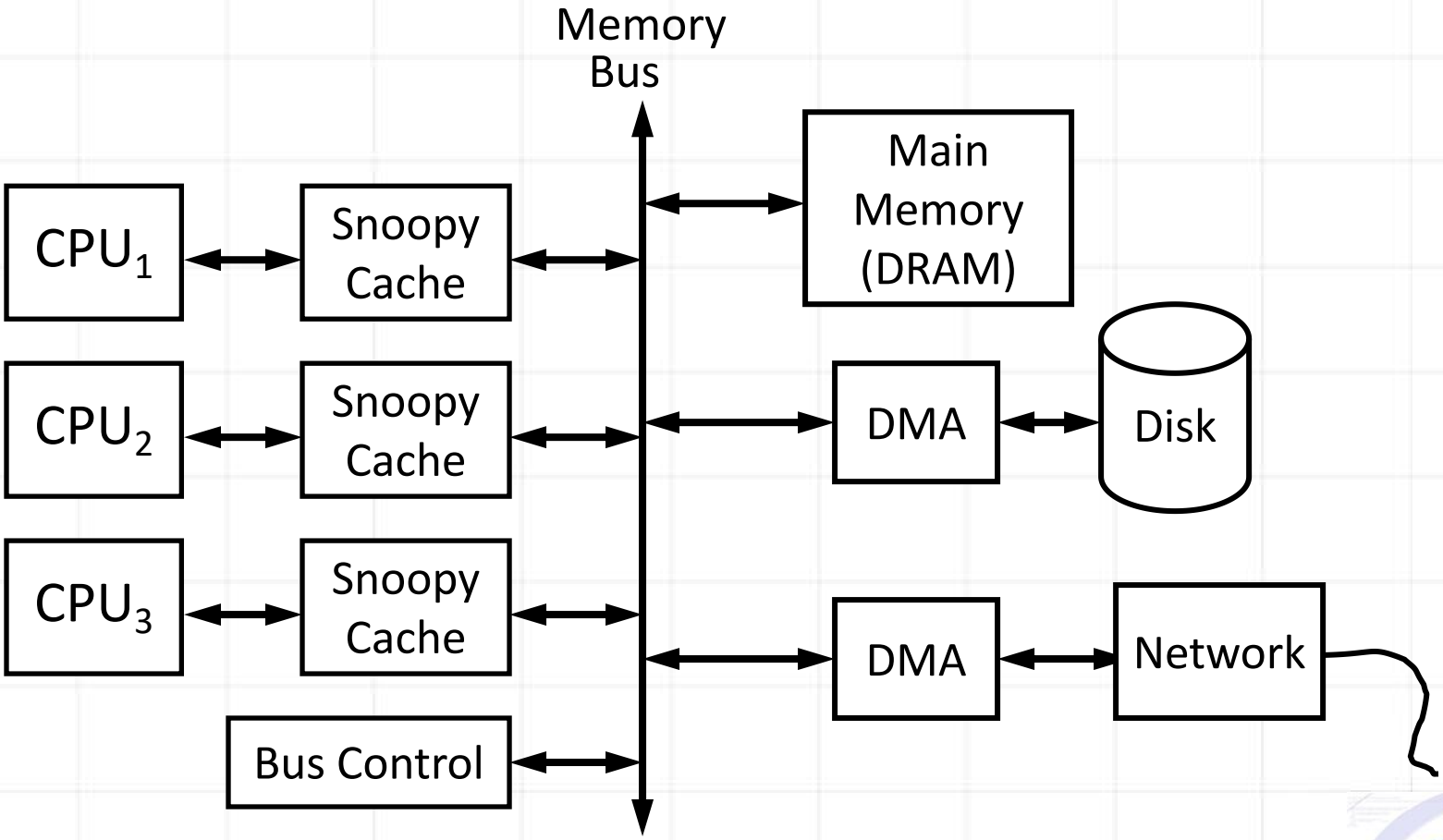


举例：写直达作废协议



8. 2. 4 写回作废监听一致性协议

写回作废协议（Snoopy）

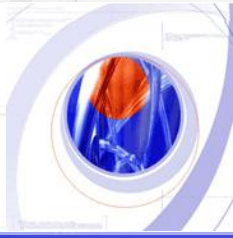


8.2.4 写回作废监听一致性协议

写回作废协议（Snoop）

1. 每个Cache块有三种状态：
   - 无效（简称I）：Cache中该块的内容为无效。
   - 共享（简称S）：该块可能处于共享状态。
     - 在多个处理器中都有副本。这些副本都相同，且与存储器中相应的块相同。
   - 已修改（简称M）：该块已经被修改过，并且还没写入存储器。

     （块中的内容是最新的，系统中唯一的最新副本）
2. Cache发送到总线上的消息主要有以下三种：
   - **RdMiss**——读不命中
   - **WtMiss**——写不命中
   - **Invalidate** ——通知其他各处理器作废其Cache中相应的副本
     - 与WtMiss的区别：Invalidate不引起调块

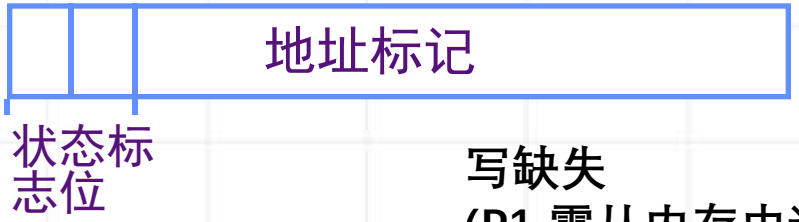


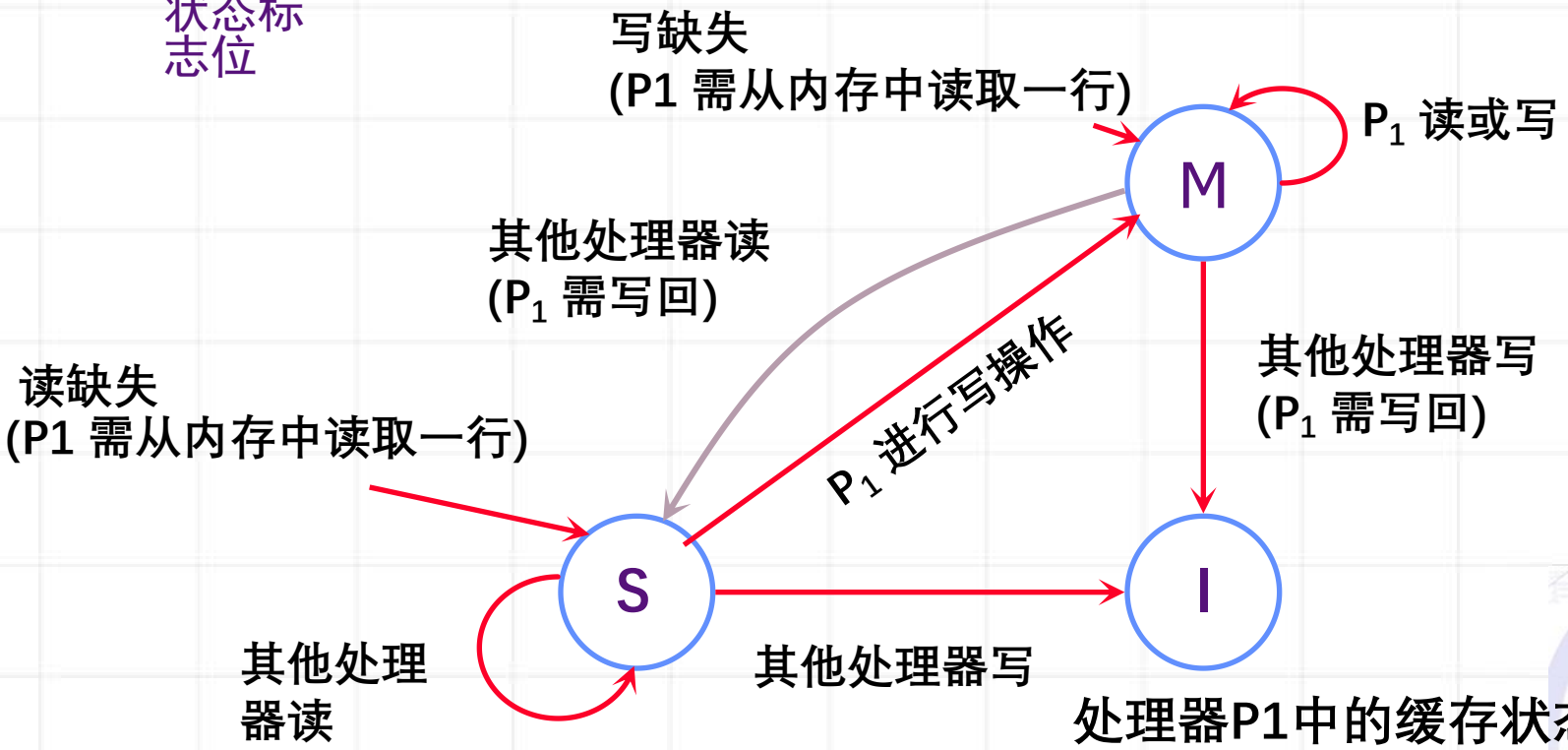
8.2.4 写回作废监听一致性协议

缓存状态转换关系图 *MSI 协议*

每个高速缓存行都有状态标志位

M: 已修改
S: 共享
I: 无效

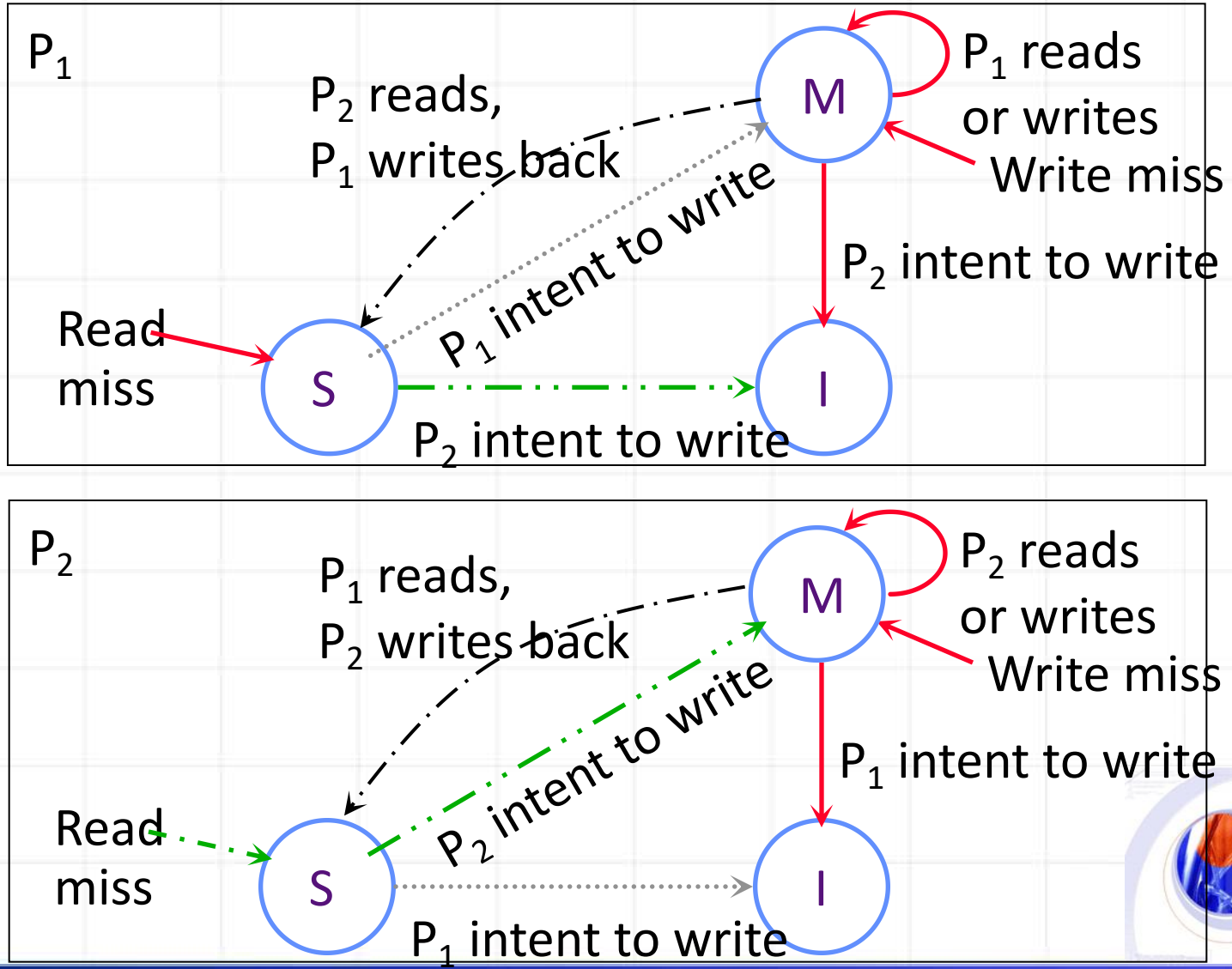




8. 2. 4 写回作废监听一致性协议

两个处理器存取同一个cache块

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

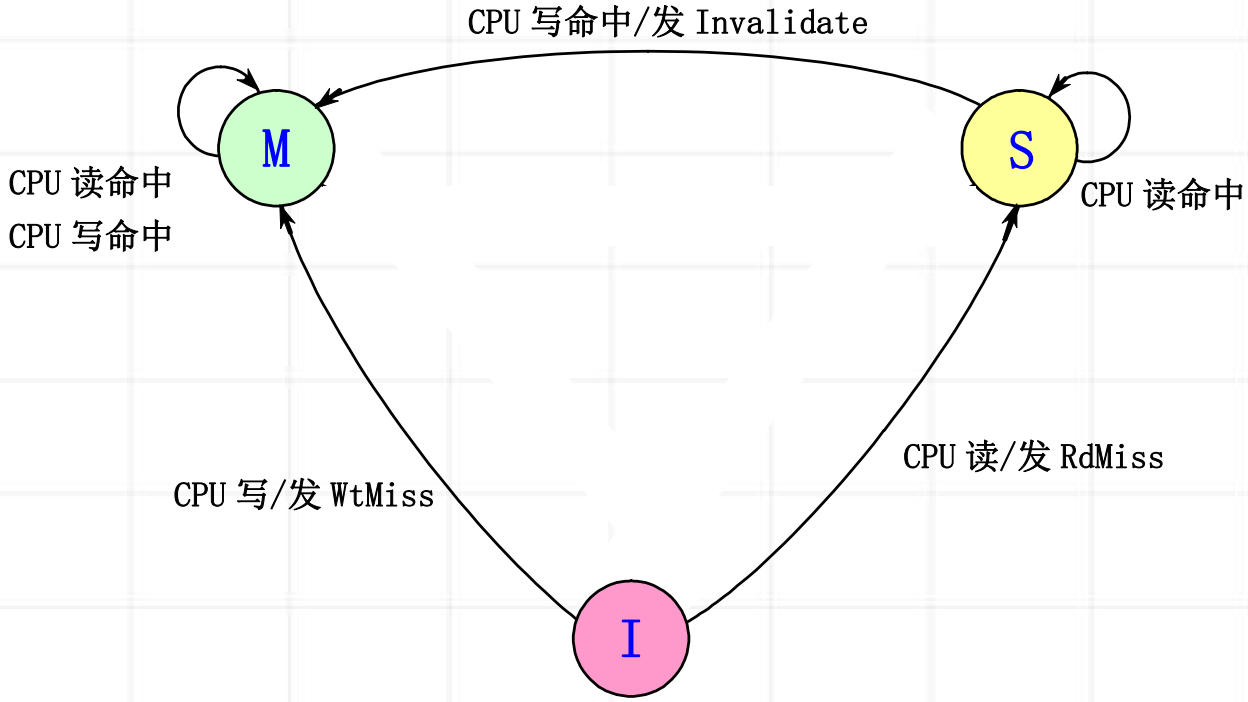


8.2.5 MSI监听协议实现

下面来讨论在各种情况下监听协议所进行的操作。

- 响应来自处理器的请求

⑩ 不发生替换的情况

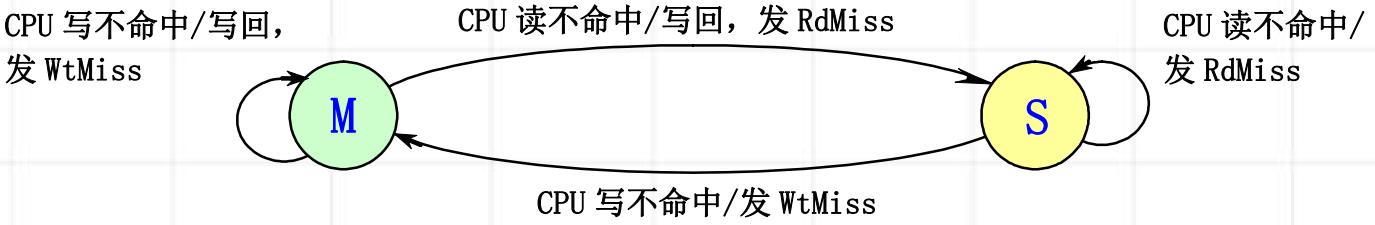


写作废协议中（采用写回法），Cache块的状态转换图

8.2.5 MSI监听协议实现

– 响应来自处理器的请求

- 发生替换的情况

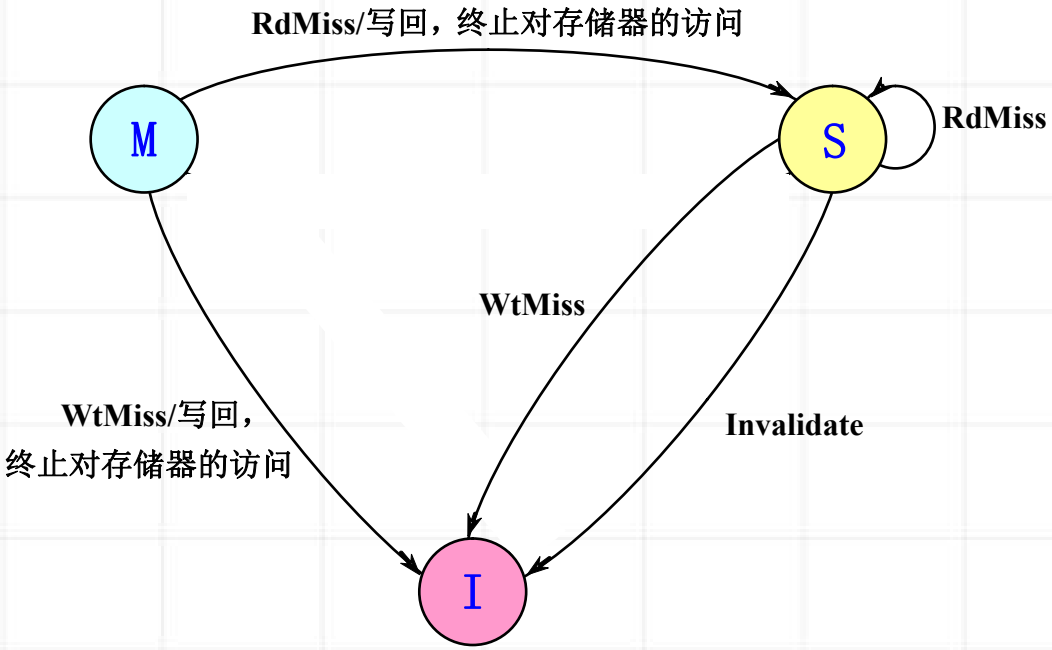


写作废协议中（采用写回法），Cache块的状态转换图

8.2.5 MSI监听协议实现

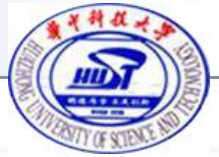
➢ 响应来自总线的请求

- 每个处理器都在监视总线上的消息和地址，当发现有与总线上的地址相匹配的Cache块时，就要根据该块的状态以及总线上的消息，进行相应的处理。



写作废协议中（采用写回法），Cache块的状态转换图

第8章 多处理器

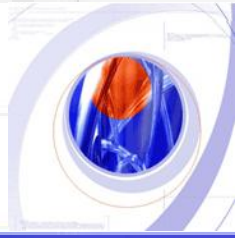


8.1 多处理器概念

8.2 一致性问题

8.3 同步

8.4 同步性能问题



8.3 同 步

同步机制通常是在硬件提供的**基本原语**基础上，通过**软件例程**来建立的。

8.3.1 基本硬件原语

在多处理机中实现同步，所需的主要功能是：

- 一组能以原子操作的方式读出并修改存储单元的硬件原语。它们都能以原子操作的方式读/修改存储单元，并指出所进行的操作是否以原子的方式进行。
- 通常情况下，用户不直接使用基本的硬件原语，原语主要供系统程序员用来编制同步库函数。■

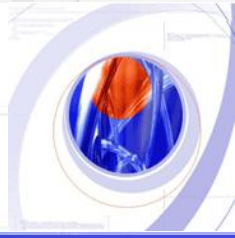
8.3.1同步原语

1. 典型操作：原子交换（atomic exchange）
   - 功能：将一个存储单元的值和一个寄存器的值进行交换。

     建立一个锁，锁值：
     - 0：表示开的（可用）
     - 1：表示已上锁（不可用）
   - 处理器上锁时，将对应于该锁的存储单元的值与存放在某个寄存器中的1进行交换。如果返回值为0，存储单元的值此时已置换为1，防止了别的进程竞争该锁。
   - 实现同步的关键：操作的原子性

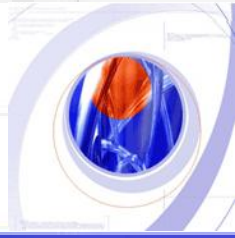
8.3.1同步原语

2. 测试并置定（test_and_set）
   - 先测试一个存储单元的值，如果符合条件则修改其值。
3. 读取并加1（fetch_and_increment）
   - 它返回存储单元的值并自动增加该值。■
4. 使用指令对
   - LL(load linked或load locked)的取指令
   - SC(store conditional)的特殊存指令



8.3.1同步原语

- 指令顺序执行：
  - 如果由LL指明的存储单元的内容在SC对其进行写之前已被其它指令改写过，则第二条指令SC执行失败；
  - 如果在两条指令间进行切换也会导致SC执行失败。
  - LL返回该存储单元初始值。
  - SC将返回一个值来指出该指令操作是否成功：
    - “1”：成功
    - “0”：不成功



8.3.1同步原语

例：实现对由R1指出的存储单元进行原子交换操作。

```
try: OR    R3, R4, R0      // R4中为交换值。把该值送入R3
     LL    R2, 0（R1）
                           // 把单元0（R1）中的值取到R2
     SC    R3, 0（R1）     // 若0（R1）中的值与R3中的值相
                              同，则置R3的值为1，否则置为0
     BEQZ  R3, try         // 存失败（R3的值为0）则转移
     MOV   R4, R2          // 将取的值送往R4
```

最终R4和由R1指向的单元值进行原子交换，在LL和SC之间如有别的处理器插入并修改了存储单元的值，SC将返回0并存入R3中，从而使这段程序再次执行。

8.3.1同步原语

- LL/SC机制的一个优点：用来构造别的同步原语

例如：构造原子操作fetch_and_increment：

```
try:    LL      R2, 0(R1)
        DADDIU  R2, R2, #1
        SC      R2, 0(R1)
        BEQZ    R2, try
```

- 指令对的实现必须跟踪地址

由LL指令指定一个寄存器，该寄存器存放着一个单元地址，这个寄存器常称为连接寄存器。

8.3.2 用一致性实现锁

- 采用多处理机的一致性机制来实现旋转锁。
- 旋转锁

处理器环绕一个锁不停地旋转而请求获得该锁。适合于这样的场合：锁被占用的时间很少，在获得锁后加锁过程延迟很小。

8.3.2 旋转锁概念

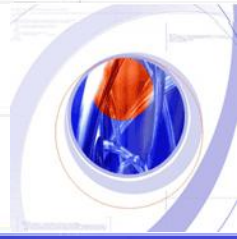
1. 无Cache一致性机制

在存储器中保存锁变量，处理器可以不断地通过一个原子操作请求使用权。

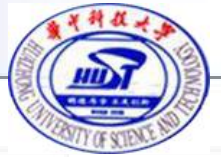
比如：利用原子交换操作，并通过测试返回值而知道锁的使用情况。释放锁的时候，处理器只需简单地将锁置为0。

例：用原子交换操作对旋转锁进行加锁，R1中存放的是该旋转锁的地址。

```
        DADDIU    R2, R0, #1
lockit: EXCH      R2, 0(R1)
        BNEZ      R2, lockit
```



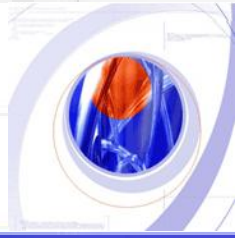
8.3.2 旋转锁概念



2. 支持Cache一致性

- 将锁调入Cache，并通过一致性机制使锁值保持一致。
- 优点：
  - 可使“环绕”的进程只对本地Cache中的锁（副本）进行操作，而不用在每次请求占用锁时都进行一次全局的存储器访问；
  - 可利用访问锁时所具有的局部性，即处理器最近使用过的锁不久又会使用。

    （减少为获得锁而花费的时间）



8.3.2 旋转锁概念

- 改进旋转锁（获得第一条好处）
  - 只对本地Cache中锁的副本进行读取和检测，直到发现该锁已经被释放。然后，该程序立即进行交换操作，去跟在其它处理器上的进程争用该锁变量。
  - 修改后的旋转锁程序：

```
lockit:  LD      R2, 0(R1)
         BNEZ    R2, lockit
         DADDIU  R2, R0, #1
         EXCH    R2, 0(R1)
         BNEZ    R2, lockit
```

  - 3个处理器利用原子交换争用旋转锁所进行的操作

8.3.3 旋转锁竞争

3个处理器利用原子交换争用旋转锁所进行的操作

步骤	处理器P0	处理器P1	处理器P2	锁的状态	总线/目录操作
1	占有锁	环绕测试 是否lock=0	环绕测试 是否lock=0	共享	无
2	将锁置为0	（收到作废命令）	（收到作废命令）	专有（P0）	P0发出对锁变量的作废消息
3		Cache不命中	Cache不命中	共享	总线/目录收到P2 Cache不命中；锁从P0写回
4		（因总线/目录忙而等待）	lock=0	共享	P2 Cache不命中被处理
5		Lock=0	执行交换， 执行Cache不命中	共享	P1 Cache不命中被处理
6		执行交换， 执行Cache不命中	交换完毕：返回0 并置lock=1	专有（P2）	总线/目录收到P2 Cache不命中；发作废消息
7		交换完毕： 返回1	进入关键程序段	专有（P1）	总线/目录处理P1 Cache不命中；写回
8		环绕测试 是否lock=0			无

8.3.3 旋转锁竞争

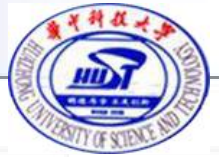
- LL／SC原语的另一个优点：读写操作明显分开

  LL命中不产生总线数据传输，这使下面代码与使用经过优化交换的代码具有相同的特点：

```
lockit:   LL      R2, 0（R1）
          BNEZ    R2, lockit
          DADDIU  R2, R0, #1
          SC      R2, 0（R1）
          BEQZ    R2, lockit
```

第一个分支形成环绕的循环体，第二个分支解决了两个处理器同时看到锁可用的情况下的争用问题。尽管旋转锁机制简单并且具有吸引力，但难以将它应用于处理器数量很多的情况。

第8章 多处理器

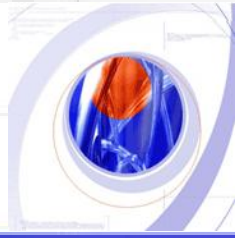


8.1 多处理器概念

8.2 一致性问题

8.3 同步

8.4 同步性能问题



8.4.1 同步性能问题

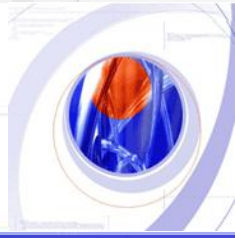
简单旋转锁不能很好地适应可缩扩性。大规模多处理机中，若所有的处理器都同时争用同一个锁，则会导致大量的争用和通信开销。

8.4.1同步性能问题

例8.3 假设某条总线上有10个处理器同时准备对同一变量加锁。如果每个总线事务处理（读不命中或写不命中）的时间是100个时钟周期，而且忽略对已调入Cache中的锁进行读写的时间以及占用该锁的时间。

（1）假设该锁在时间为0时被释放，并且所有处理器都在旋转等待该锁。问：所有10个处理器都获得该锁所需的总线事务数目是多少？

（2）假设总线是非常公平的，在处理新请求之前，要先全部处理好已有的请求。并且各处理器的速度相同。问：处理10个请求大概需要多少时间？



8.4.1同步性能问题

解 当i个处理器争用锁的时候，它们都各自完成以下操作序列，每一个操作产生一个总线事务：

- 访问该锁的i个LL指令操作
- 试图占用该锁（并上锁）的i个SC指令操作
- 1个释放锁的存操作指令

因此对于i个处理器来说，一个处理器获得该锁所要进行的总线事务的个数为2i+1。

由此可知，对n个处理器，总的总线事务个数为：

$$\sum_{i=1}^{n}(2i+1)=n(n+1)+n=n^2+2n$$

对于10个处理器来说，其总线事务数为120个，需要12000个时钟周期。

8.4.2 栅栏同步

- 本例中问题的根源：锁的争用、对锁进行访问的串行性以及总线访问的延迟。
- 旋转锁的主要优点：总线开销或网络开销比较低，而且当一个锁被同一个处理器重用时具有很好的性能。

1. 如何用旋转锁来实现一个常用的高级同步原语：栅栏
   - 栅栏强制所有到达该栅栏的进程进行等待，直到全部的进程到达栅栏，然后释放全部的进程，从而形成同步。

8.4.2 栅栏同步

- 栅栏的典型实现

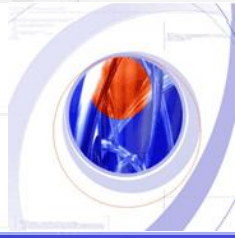
  用两个旋转锁：

  - 用来保护一个计数器，它记录已到达该栅栏的进程数；
  - 用来封锁进程直至最后一个进程到达该栅栏。

- 一种典型的实现

  其中：

  - lock和unlock提供基本的旋转锁
  - 变量count记录已到达栅栏的进程数
  - total规定了要到达栅栏的进程总数



8.4.2 栅栏同步

```
lock (counterlock) ;                    //<确保更新的原子性<
    if (count==0) release=0;            //<第一个进程则重置release
    count=count+1;                      //<到达进程数加1<
    unlock (counterlock) ;              //<释放锁<
    if (count==total) {                 //<进程全部到达<
        count=0;                        //<重置计数器<
        release=1;                      //<释放进程<
        }
    else {                              //<还有进程未到达<
        spin (release=1) ;              //<等待别的进程到达<
        }
```

8.4.2 栅栏同步

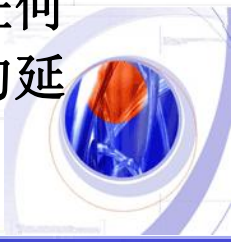
- 对counterlock加锁保证增量操作的原子性。
- release用来封锁进程直到最后一个进程到达栅栏。
- spin（release=1）使进程等待直到全部的进程到达栅栏。

➢ 实际情况中会出现的问题

栅栏通常是在循环中使用，从栅栏释放的进程运行一段后又会再次返回栅栏，这样有可能出现某个进程永远离不开栅栏的状况(它停在旋转操作上)。

- 一种解决方法

  当进程离开栅栏时进行计数（和到达时一样），在上次栅栏使用中的所有进程离开之前，不允许任何进程重用并初始化本栅栏。但这会明显增加栅栏的延迟和竞争。



8.4.2 栅栏同步

- 另一种解决办法
  - 采用sense_reversing栅栏，每个进程均使用一个私有变量local_sense，该变量初始化为1。
  - sense_reversing栅栏的代码

优缺点：使用安全，但性能比较差。

对于10个处理器来说，当同时进行栅栏操作时，如果忽略对Cache的访问时间以及其它非同步操作所需的时间，则其总线事务数为204个，如果每个总线事物需要100个时钟周期，则总共需要20400个时钟周期。

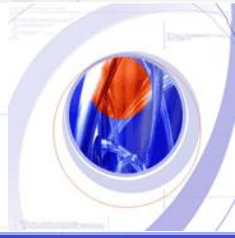
8.4.2 栅栏同步

```
local_sense=! local_sense;          //<local-sense取反<
    lock (counterlock) ;            //<确保更新的原子性<
    count++;                        //<到达进程数加1<
    unlock (counterlock) ;          //<释放锁<
    if (count==total) {             //<进程全部到达<
        count=0;                    //<重置计数器<
        release=local_sense;        //<释放进程<
        }
    else {                          //<还有进程未到达<
        spin (release==local_sense) ; //<等待信号<
        }
```

8.4.2 栅栏同步

2. 当竞争不激烈且同步操作较少时，我们主要关心的是一个同步原语操作的延迟。
   - 即单个进程要花多长时间才完成一个同步操作。
   - 基本的旋转锁操作可在两个总线周期内完成：
     - 一个读锁
     - 一个写锁

   我们可用多种方法改进，使它在单个周期内完成操作。
3. 同步操作最严重的问题：进程进行同步操作的串行化。它大幅度地增加了完成同步操作所需要的时间。



本章习题

- 10.6