

10.1 引言

1. 单处理机系统结构正在走向尽头？

2. 多处理机正起着越来越重要的作用。近几年来，人们确实开始转向了多处理机。

- Intel于2004年宣布放弃了其高性能单处理器项目，转向多核（multi-core）的研究和开发。
- IBM、SUN、AMD等公司
- 并行计算机应用软件已有了稳定的发展。
- 充分利用商品化微处理器所具有的高性能价格比的优势。

3. 本章重点：中小规模的计算机(处理器的个数 <32)
(多处理机设计的主流)

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini,
*describing Intel's future direction at the
Intel Developers Forum in 2005*

The Teraflops Research Chip

Frequency	Voltage	Power	Aggregate Bandwidth	Performance
3.16 GHz	0.95 V	62W	1.62 Terabits/s	1.01 Teraflops
5.1 GHz	1.2 V	175W	2.61 Terabits/s	1.63 Teraflops
5.7 GHz	1.35 V	265W	2.92 Terabits/s	1.81 Teraflops

10.1.1 并行计算机系统结构的分类

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。

多处理机系统是MIMD计算机的一种实现类型，也是目前已经商品化的MIMD的唯一形式。

多处理机系统由多台处理机连接而成，它们能够并行执行独立的程序模块，并且相互通信和同步，以实现作业、任务级的并行。

MIMD计算机与SIMD计算机的主要区别，在于SIMD只能在同一时刻做多件相同的事情，而MIMD却可以在同一时刻做多件相同或不同的事情（多指令流所致），所以求解同一个问题时采用MIMD将能实现更大比例的并行操作，即处理效率更高。

从并行处理的级别看，SIMD是数据级并行处理，流水线是指令级并行处理，MIMD是任务级并行处理。

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）

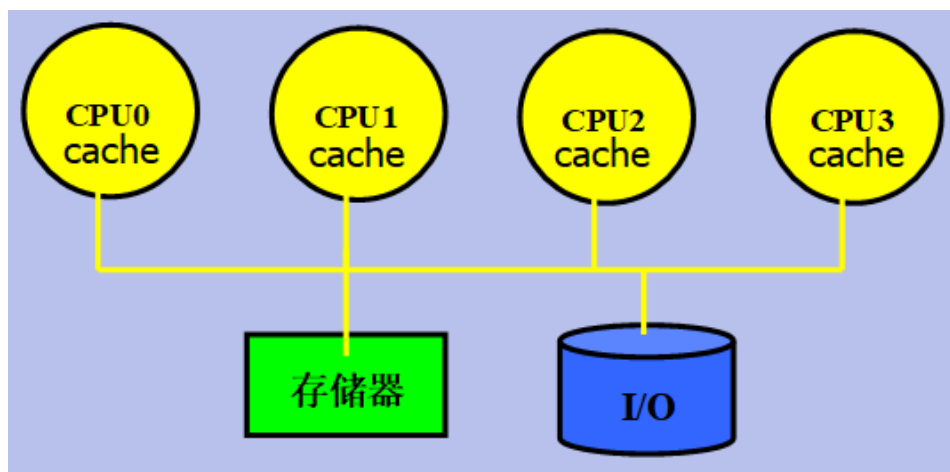
➤ 集中式共享存储器结构

- 最多由几十个处理器构成。各处理器共享集中式物理存储器。

这类机器有时被称为

SMP机器 (Symmetric shared-memory MultiProcessor) .

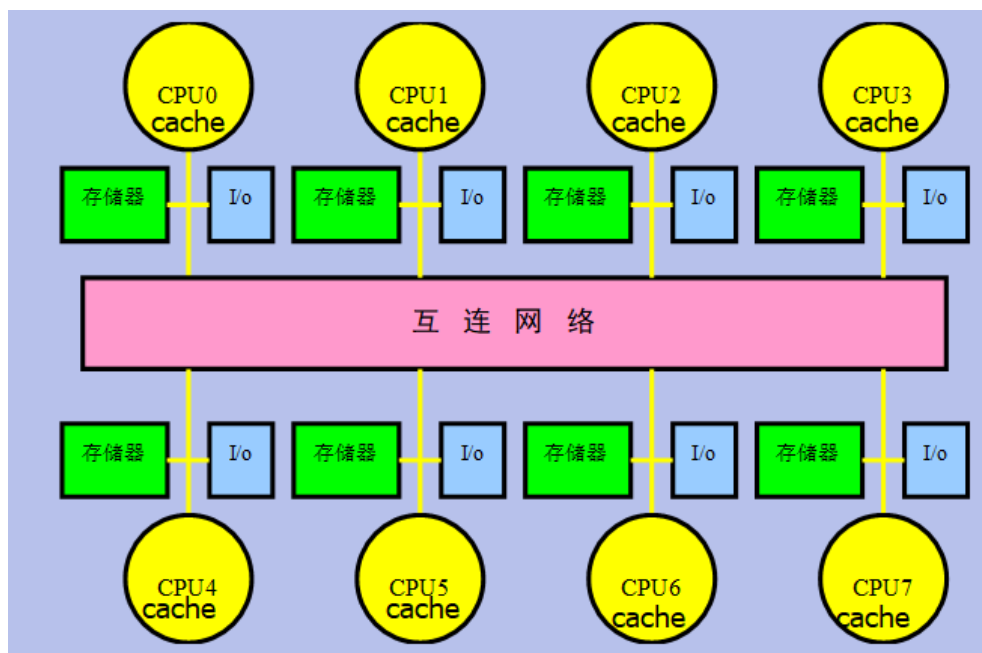
UMA机器 (Uniform Memory Access)



对称式共享存储器多处理机的基本结构

➤ 分布式存储器多处理机

- ❑ 存储器在物理上是分布的。支持较大规模多处理机系统
- ❑ 每个结点包含：
 - 处理器, CACHE
 - 存储器
 - I / O
 - 互连网络接口
- ❑ 在许多情况下，分布式存储器结构优于集中式共享存储器结构。



- 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。
- **最主要缺点**
 - 处理器间的通信较复杂，且各处理器间访问延迟较大。
- **簇：超级结点**
 - 每个结点内包含个数较少（例如**2~8**）的处理器；
 - 处理器之间可采用另一种互连技术（例如总线）相互连接形成簇。

10.1.2 存储器系统结构和通信机制

1. 两种存储器系统结构

➤ 共享地址空间

- ❑ 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。
- ❑ 任何一个处理器可以访问该共享空间中的任何一个单元（如果它具有访问权），而且不同处理器上的同一个物理地址指向的是同一个存储单元。

❑ 这类计算机被称为

分布式共享存储器系统 (DSM: Distributed Shared-Memory)

NUMA机器 (NUMA: Non-Uniform Memory Access)

- 把每个结点中的存储器编址为一个独立地址空间，不同结点中的地址空间是相互独立的。
- ❑ 整个系统的地址空间由多个独立的地址空间构成
- ❑ 每个结点中的存储器只能由本地处理器访问，远程处理器不能直接对其进行访问。
- ❑ 每一个处理器-存储器模块实际上是一台单独计算机
- ❑ 现在的这种机器多以集群的形式存在

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
- 处理器间通过load/store指令对相同存储器地址读/写来实现

➤ 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，处理器间通过发送消息通信，这些消息请求进行某些操作或者传送数据。

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后，执行相应操作或代替远程处理器进行访问，并发送应答消息返回结果。

□ 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

数据发送方知道别处理器需要数据，通信也可从数据发送方开始，数据可不经请求就直接送往数据接受方。

3. 不同通信机制的优点

➤ 共享存储器通信

- ❑ 与常用对称式多处理机的通信机制兼容。
- ❑ 易于编程，在简化编译器设计方面有优势。
- ❑ 采用大家熟悉的共享存储器模型开发应用程序，把重点放到解决对性能影响较大的数据访问上。
- ❑ 通信数据量较小时，通信开销较低，带宽利用较好。
- ❑ 可通过Cache技术减少远程通信频度，减少通信延迟以及对共享数据访问冲突。

➤ 消息传递通信机制

- ❑ 硬件较简单。
- ❑ 显式通信，更易搞清何时发生通信以及通信开销是多少。
- ❑ 显式通信可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、性能更高的并程序。
- ❑ 同步很自然地与发送消息相关联，能减少不当同步带来错误的可能性。

10.1.3 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

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

10.1.3 并行处理面临的挑战

多处理机加快运算速度的基本原理是并行计算，即把一个程序分成 n 等份交给 n 个处理机同时执行，但在实现上难度较大。主要困难有2个：

(1) 程序中总有一定比例的不可并行部分，如I/O操作；

(2.1) 划分开的各部分之间需要通信（同步、互斥等），分得越细通信越频繁，而单处理机算法中几乎不需通信（即使进程间需通信，时间开销也小到可忽略）；

(2.2) 多处理机分类中的多计算机系统的通信方式是机外传输，每次时间开销比机内传输大成百上千倍。

对算法的要求：

(1) 较高的可并行化比例；

(2) 较低的进程间通信量。

只有满足这2个要求的问题才适合在多处理机系统求解。

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10.1 假设有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

2. 第二个挑战：多处理机中远程访问的延迟较大

- 在现有机器中，处理器间数据通信约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

- 在几种不同共享存储器并行计算机中远程访问一个字的典型延迟

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

例10.2 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？

解 有0.2%远程访问的机器的实际CPI为：

$$\begin{aligned}\text{CPI} &= \text{基本CPI} + \text{远程访问率} \times \text{远程访问开销} \\ &= 0.5 + 0.2\% \times \text{远程访问开销}\end{aligned}$$

远程访问开销为：

$$\begin{aligned}\text{远程访问时间/时钟周期时间} &= 200\text{ns}/0.5\text{ns} = 400\text{个时钟周期} \\ \therefore \text{CPI} &= 0.5 + 0.2\% \times 400 = 1.3\end{aligned}$$

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

➤ 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

4. 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减少。

10.2 对称式共享存储器系统结构

- 多个处理器通过共享总线共享一个存储器
- 大容量、多级Cache降低了对内存带宽和总线带宽的要求

2005, AMD&Intel two processor for server

2006, Sun T1 eight processpr multicore

- 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
 - 共享数据则是供多个处理器使用
- 共享数据进入Cache产生了一个新的问题

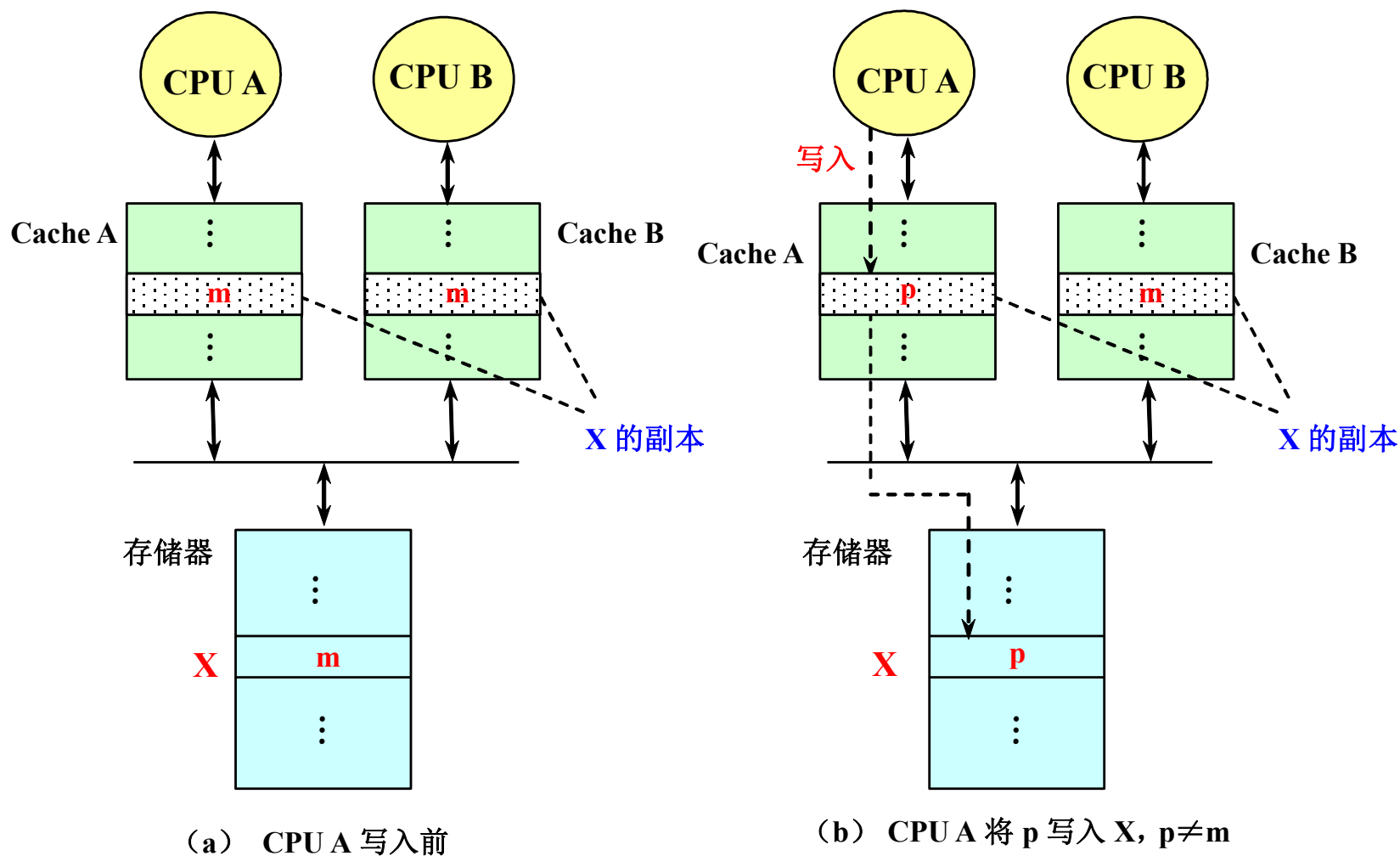
Cache的一致性问题

10.2.1 多处理机Cache一致性

1. 多处理机的Cache一致性问题

- 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。

例 由两个处理器（A和B）读写引起的Cache一致性问题



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

➤ 存储器是一致的，当满足以下条件

1. 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。

2. 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。
3. 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来看顺序都是相同的。（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。即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。

10.2.2 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

减少了对远程共享数据的访问延迟，也减少了对共享存储器带宽的要求。

- 共享数据的复制

不仅减少了访问共享数据的延迟，也减少了访问共享数据所产生的冲突。

一般情况下，小规模多处理机是采用硬件的方法来实现Cache的一致性。

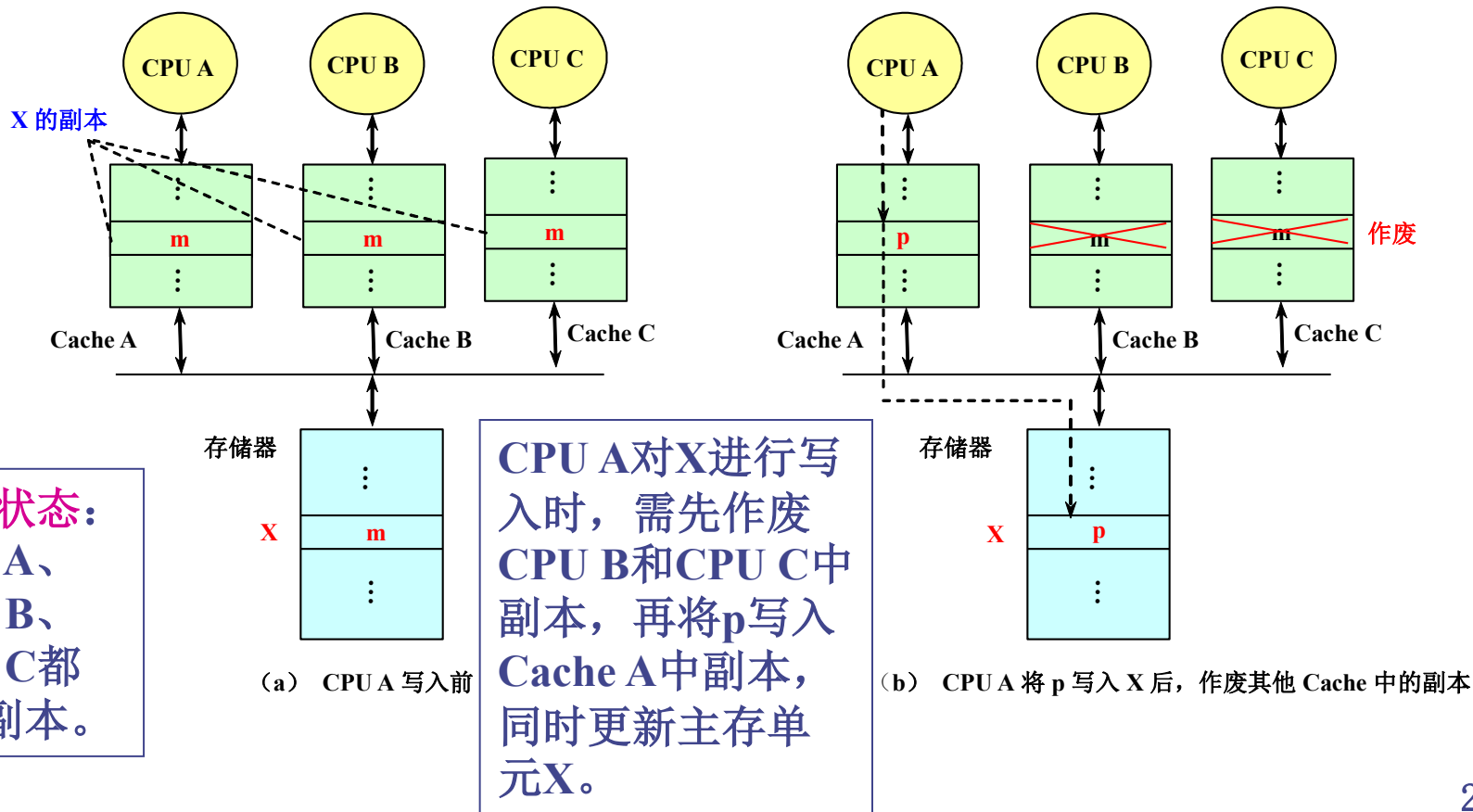
1. **Cache一致性协议**：在多个处理器中用来维护一致性的协议。

- **关键**：跟踪记录共享数据块的状态
- **两类协议**（采用不同技术跟踪共享数据状态）
 - **监听式协议**（snooping）
 - 每个Cache除了包含物理存储器中块的数据拷贝外，也保存着各块的共享状态信息。
 - 所有Cache都可通过某种广播介质访问，所有Cache控制器监听总线来判断它们是否有总线上请求的数据块。
 - **目录式协议**（directory）
 - 物理存储器中数据块的共享状态被保存在一个称为目录的地方。开销稍大于前者，但可用于实现更大规模的多处理机。

2. 采用两种方法来解决Cache一致性问题。

- **写作废协议：** 在处理器对某数据项写入之前，保证它拥有对该数据项的唯一访问权。（作废其它的副本）

例 监听总线、写作废协议举例（采用写直达法）



➤ 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。

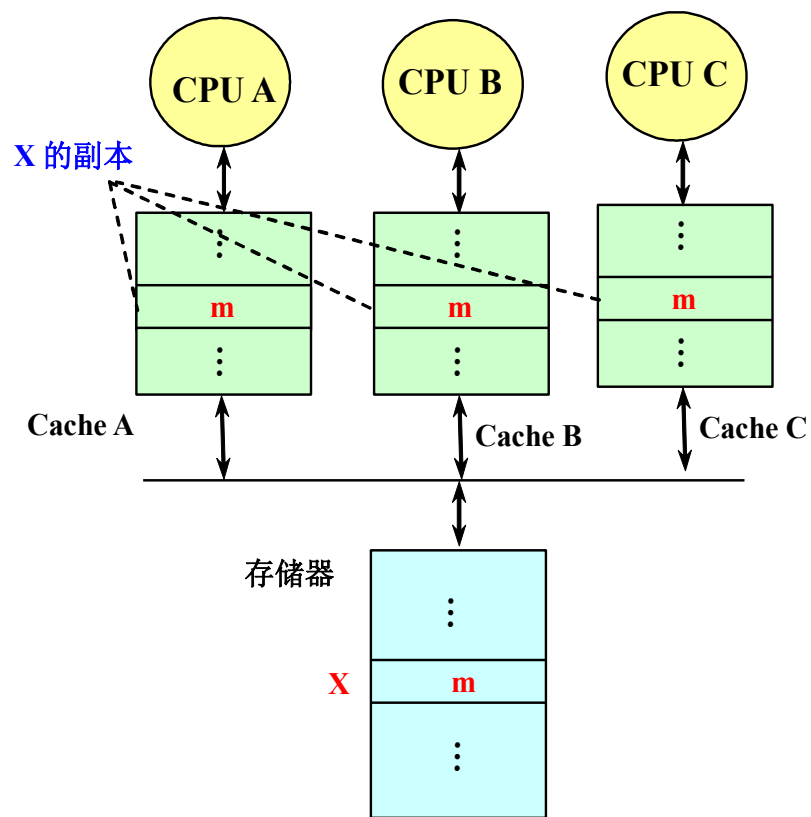
例 监听总线、写更新协议举例（采用写直达法）

假设：3个Cache都有X的副本。

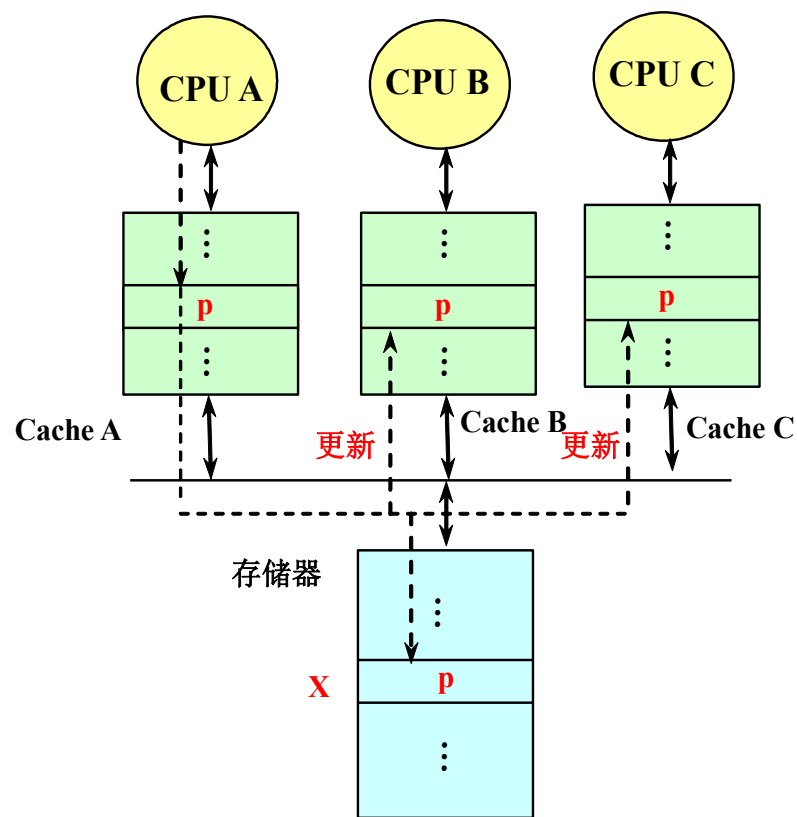
当CPU A将数据p写入Cache A中的副本时，将p广播给所有的Cache，这些Cache用p更新其中的副本。

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。

10.2 对称式共享存储器系统结构



(a) CPU A 写入前



(b) CPU A 将 p 写入 X 后，更新其他 Cache 中的副本

- 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时，写更新协议需多次写广播操作，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时，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块的，写更新则是针对字（或字节）。
 - 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。
- 后面讲述关注写作废协议

10.2.3 监听协议的实现

1. 监听协议的基本实现技术

- 实现写作废协议的关键：使用广播介质执行作废（假定采用的是总线）
 - 当一个处理器Cache响应本地CPU访问时涉及全局操作，Cache控制器需取得总线访问权后，在总线上发相应消息。
 - 所有处理器都一直在监听总线，检测总线上的地址是否在它们的Cache中。若在，则相应处理。
- 写操作的串行化：由总线实现（获取总线控制权的顺序性）

2. Cache发送到总线上的消息主要有以下两种：

- **RdMiss**——读不命中
 - **WtMiss**——写不命中
- 需要通过总线找到相应数据块的最新副本，然后调入本地Cache中。
- **写直达Cache**：因为所有写入的数据都同时被写回主存，所以从主存中总可以取到其最新值。
 - **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。
(后面的讨论中，只考虑写回法Cache)

- 有的监听协议还增设了 **Invalidate** 消息，用来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
 - Cache 的 **标识**（**tag**）可直接用来实现监听。
 - **修改位**—最新副本（写回法）
 - 作废一个块只需将其 **有效位置** 为无效。
 - 给每个 Cache 块增设一个 **共享位**
 - 为 “**1**”：该块是被多个处理器所共享
 - 为 “**0**”：仅被某个处理器所独占
- 块的拥有者**：拥有该数据块的唯一副本的处理器。

3. 监听协议举例

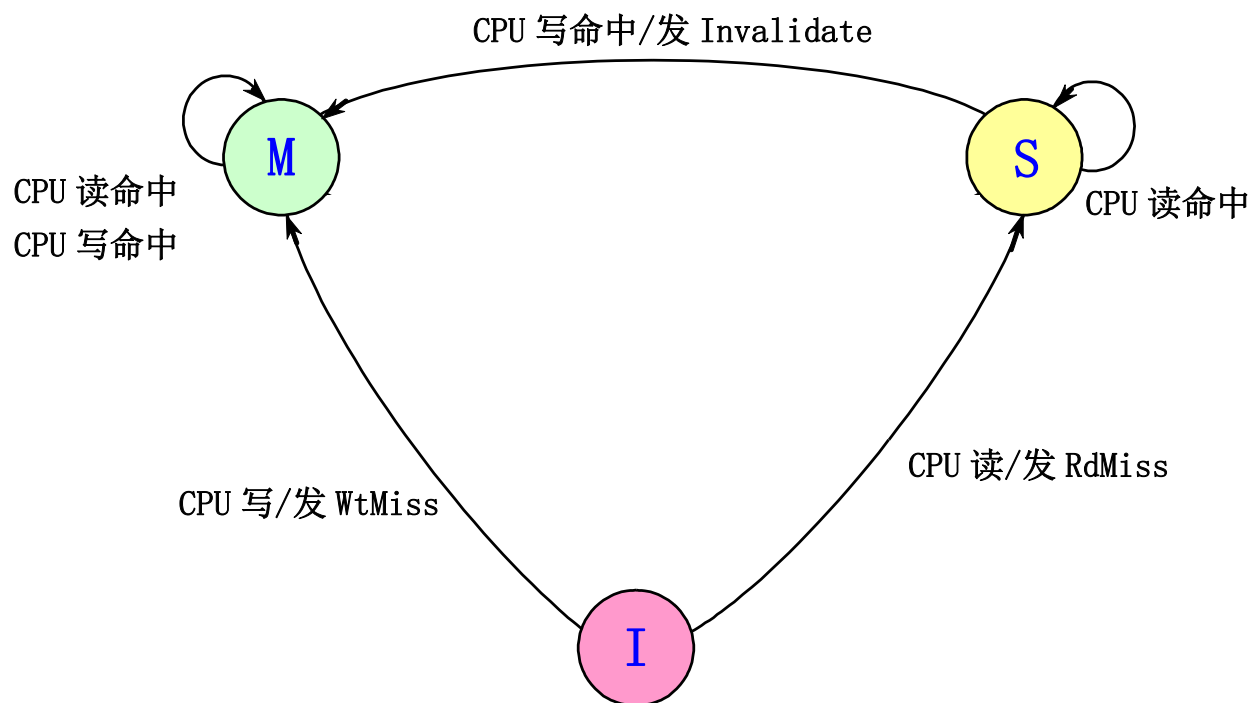
- 在每个结点内嵌入一个有限状态控制器。
 - 该控制器根据来自处理器或总线的请求以及Cache块的状态，做出相应的响应。
- 每个数据块的状态取以下3种状态中的一种：
 - 无效（简称I）：Cache中该块的内容为无效。
 - 共享（简称S）：该块可能处于共享状态。
 - 在多个处理器中都有副本。这些副本都相同，且与存储器中相应的块相同。
 - 已修改（简称M）：该块已经被修改过，并且还没写入存储器。

（块中的内容是最新的，系统中唯一的最新副本）

下面来讨论在各种情况下监听协议所进行的操作。

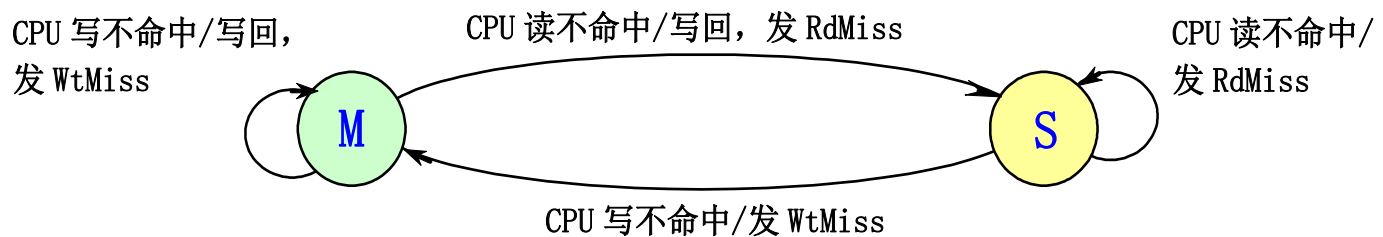
➤ 响应来自处理器的请求

□ 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

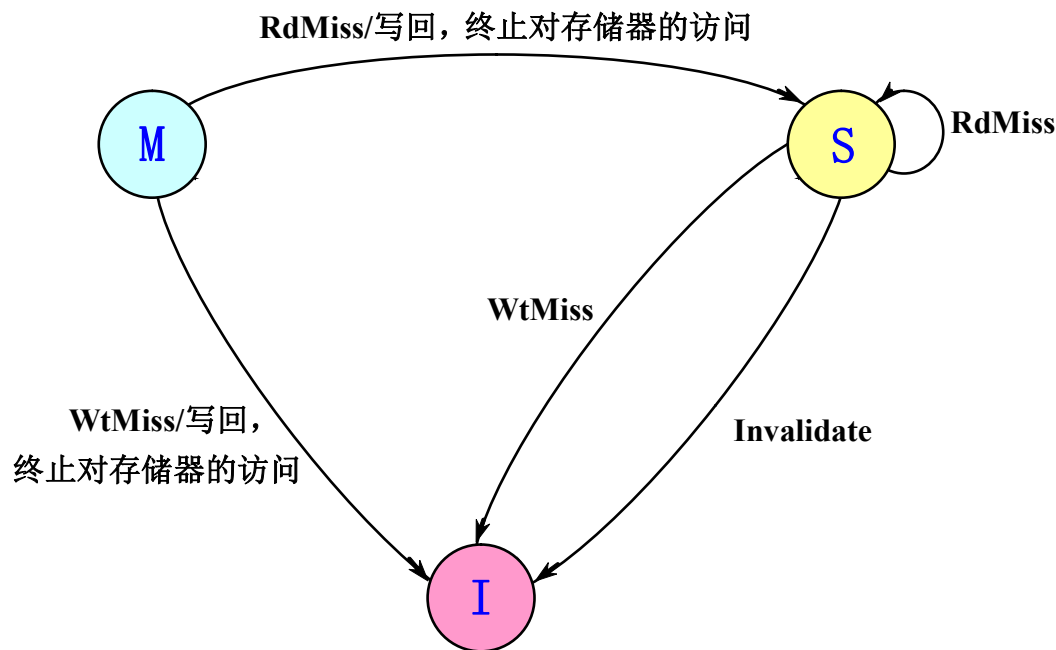
□ 发生替换的情况



写作废协议中（采用写回法），Cache块的状态转换图

➤ 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

10.3 分布式共享存储器系统结构

10.3.1 目录协议的基本思想

- 广播和监听的机制使得监听一致性协议的可扩展性很差。
- 寻找替代监听协议的一致性协议。

（采用目录协议）

1. 目录协议

- **目录：**一种集中的数据结构。对于存储器中的每一个可以调入Cache的数据块，在目录中设置一条目录项，用于记录该块的状态以及哪些Cache中有副本等相关信息。

- **特点:**

对于任何一个数据块，都可以快速地在唯一的一个位置中找到相关的信息。这使一致性协议避免了广播操作。

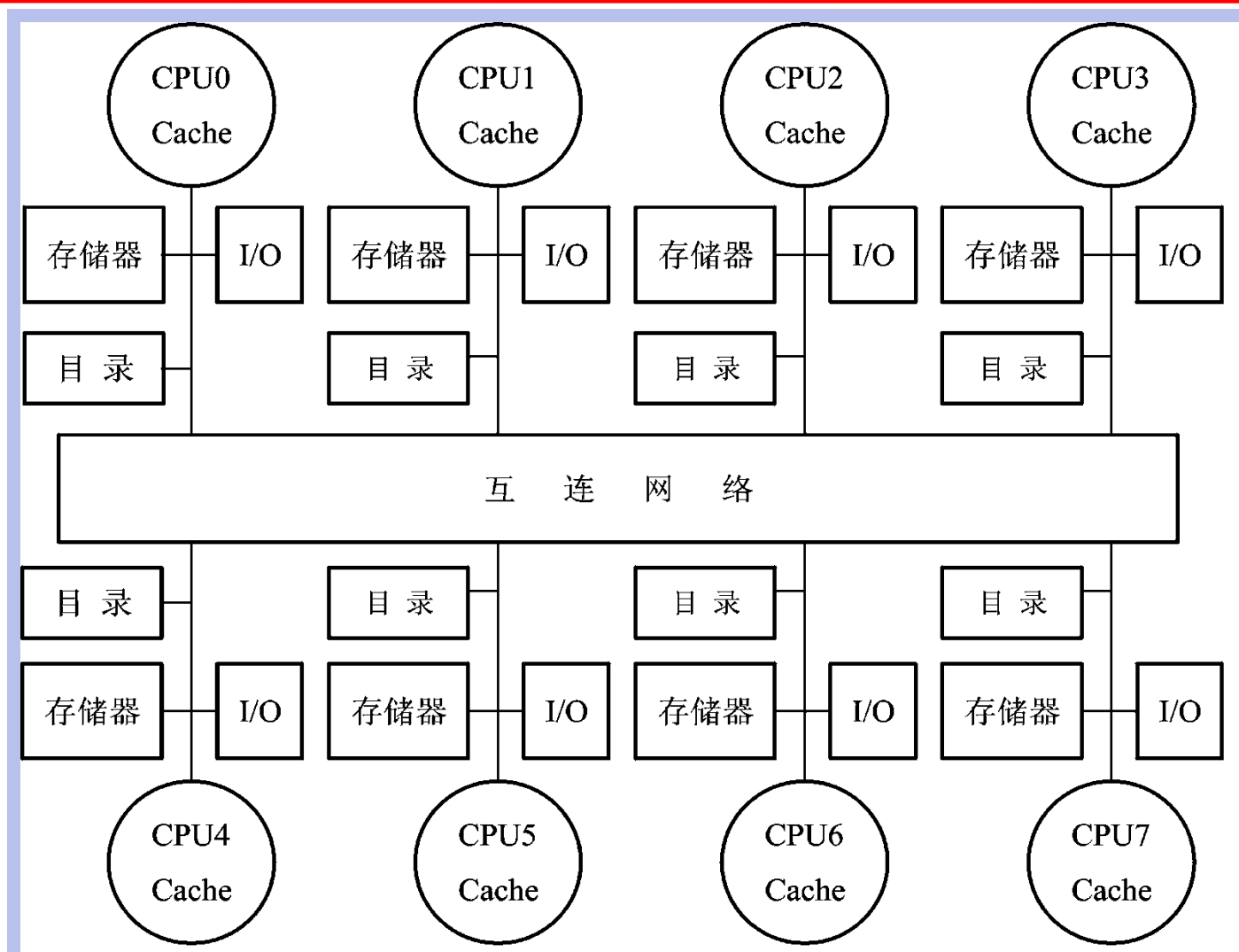
- **位向量:** 记录哪些Cache中有副本。

- 每一位对应于一个处理器。
- 长度与处理器的个数成正比。
- 由位向量指定的处理机的集合称为**共享集S**。

- **分布式目录**

- 目录与存储器一起分布到各结点中，从而对于不同目录内容的访问可以在不同的结点进行。

□ 对每个结点增加目录后的分布式存储器多处理机



- 目录法最简单的实现方案：对于存储器中每一块都在目录中设置一项。目录中的信息量与 $M \times N$ 成正比。

其中：

- M ：存储器中存储块的总数量
- N ：处理器的个数
- 由于 $M=K \times N$ ， K 是每个处理机中存储块的数量，所以如果 K 保持不变，则目录中的信息量就与 N^2 成正比。

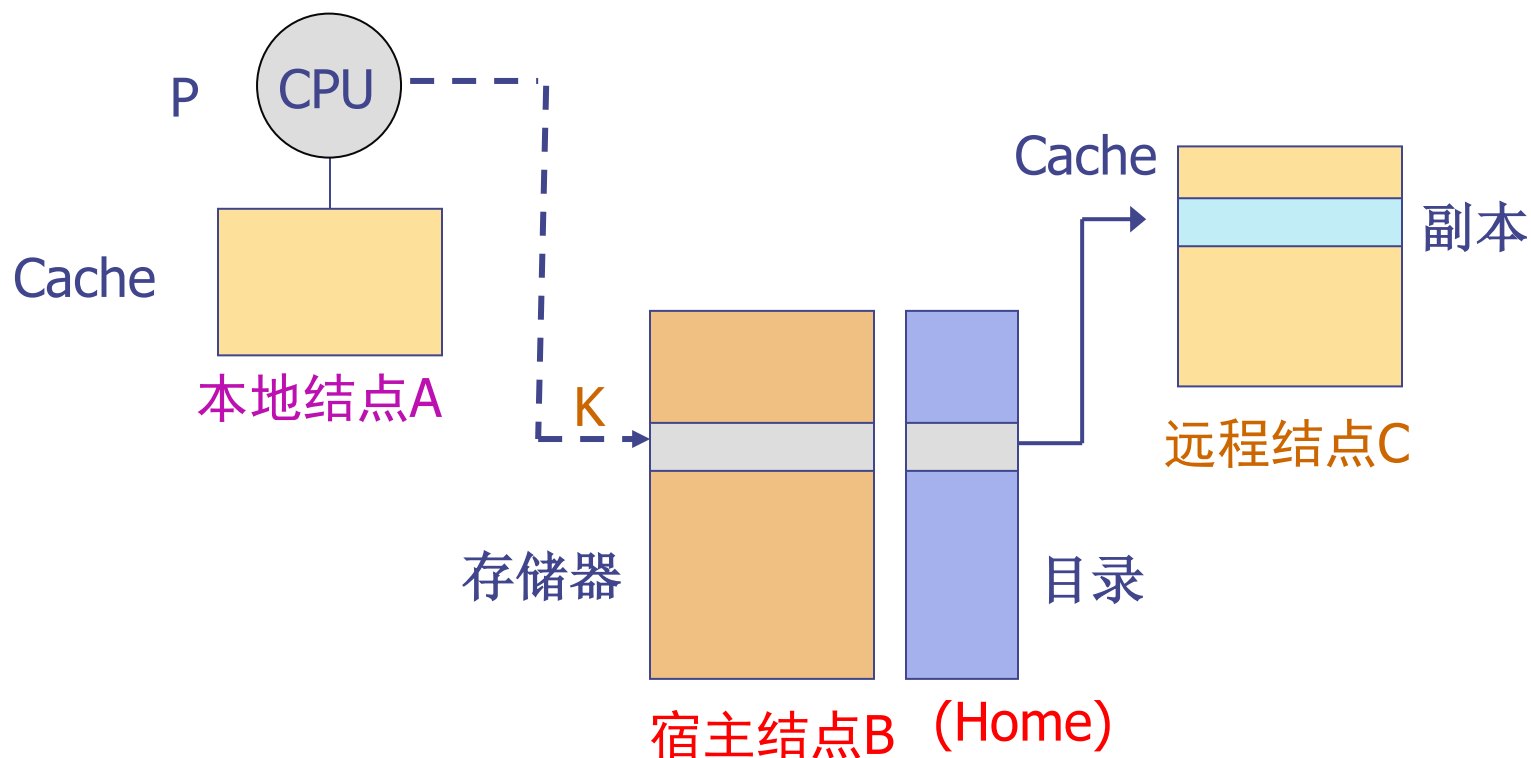
2. 在目录协议中，存储块的状态有3种：

- **未缓冲**：该块尚未被调入Cache。所有处理器的Cache中都没有这个块的副本。
- **共享**：该块在一个或多个处理机上有这个块的副本，且这些副本与存储器中的该块相同。
- **独占**：仅有一个处理机有这个块的副本，且该处理机已经对其进行了写操作，所以其内容是最新的，而存储器中该块的数据已过时。

这个处理机称为该**块的拥有者**。

3. 本地结点、宿主结点以及远程结点的关系

- **本地结点：**发出访问请求的结点
- **宿主结点：**包含所访问的存储单元及其目录项的结点
- **远程结点：**包含Cache block的副本。



4. 在结点之间发送的消息

➤ 本地结点发给宿主结点（目录）的消息

P: 发出请求的处理机编号, K: 所要访问的地址

□ RdMiss (P, K)

处理机P读取地址为A的数据时不命中, 请求宿主结点提供数据（块）, 并要求把P加入共享集。

□ WtMiss (P, K)

处理机P对地址A进行写入时不命中, 请求宿主结点提供数据, 并使P成为所访问数据块的独占者。

□ Invalidate (K)

请求向所有拥有相应数据块副本（包含地址K）的远程Cache发Invalidate消息, 作废这些副本。

➤ 宿主结点（目录）发送给远程结点的消息

□ Invalidate (K)

作废远程Cache中包含地址K的数据块。

□ Fetch (K)

从远程Cache中取出包含地址K的数据块，并送到宿主结点。把远程Cache中那个块的状态改为“共享”。

□ Fetch&Inv (K)

从远程Cache中取出包含地址K的数据块，并送到宿主结点。然后作废远程Cache中的那个块。

➤ 宿主结点发送给本地结点的消息

□ DReply (D)

- D表示数据内容。
- 把从宿主存储器获得的数据返回给本地Cache。

➤ 远程结点发送给宿主结点的消息

□ WtBack (K, D)

- 把远程Cache中包含地址K的数据块写回到宿主结点中，该消息是远程结点对宿主结点发来的“取数据”或“取/作废”消息的响应。

➤ 本地结点发送给被替换块的宿主结点的消息

□ MdSharer (P, K)

用于当本地Cache中需要替换一个包含地址K的块、且该块未被修改过。这个消息发给该块宿主结点，请求它将P从共享集中删除。如果删除后共享集为空，则宿主结点还要将该块状态改为“未缓存”（U）。

□ WtBack2 (P, K, D)

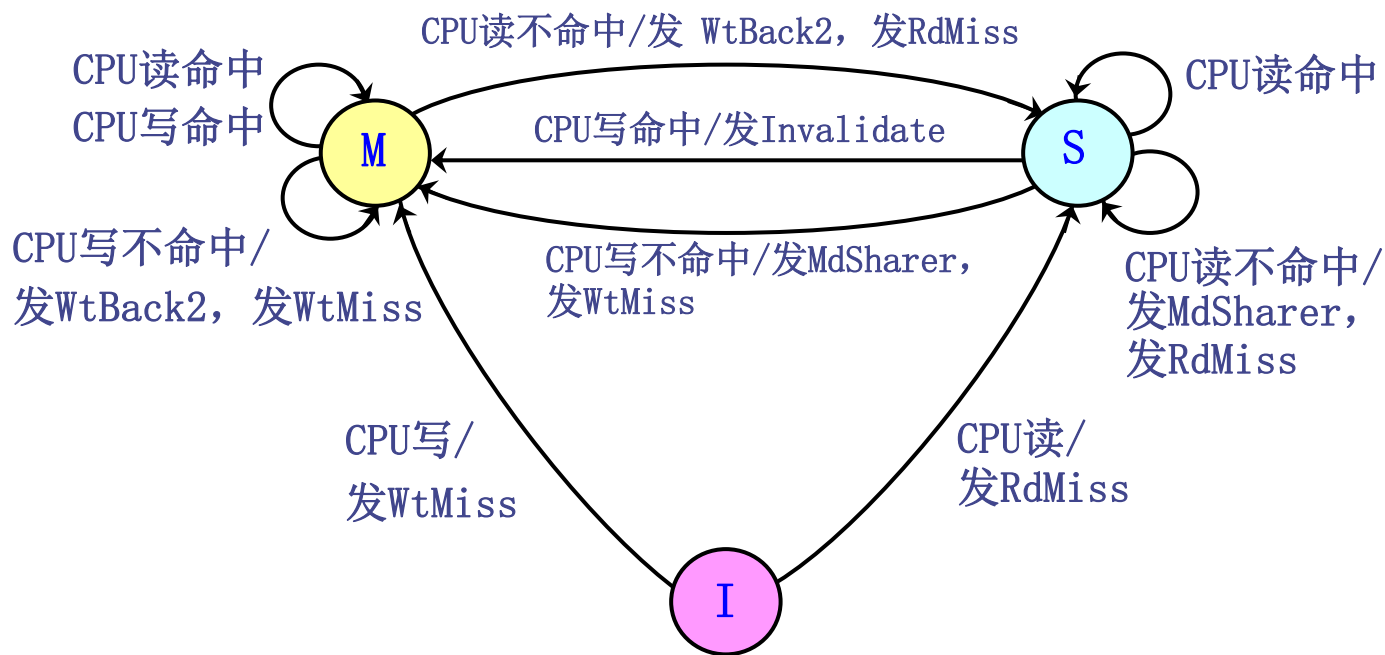
用于当本地Cache中需要替换一个包含地址K的块、且该块已被修改过。这个消息发给该块宿主结点，完成两步：①写回该块；②进行与MdSharer相同操作。

10.3.2 目录协议实例

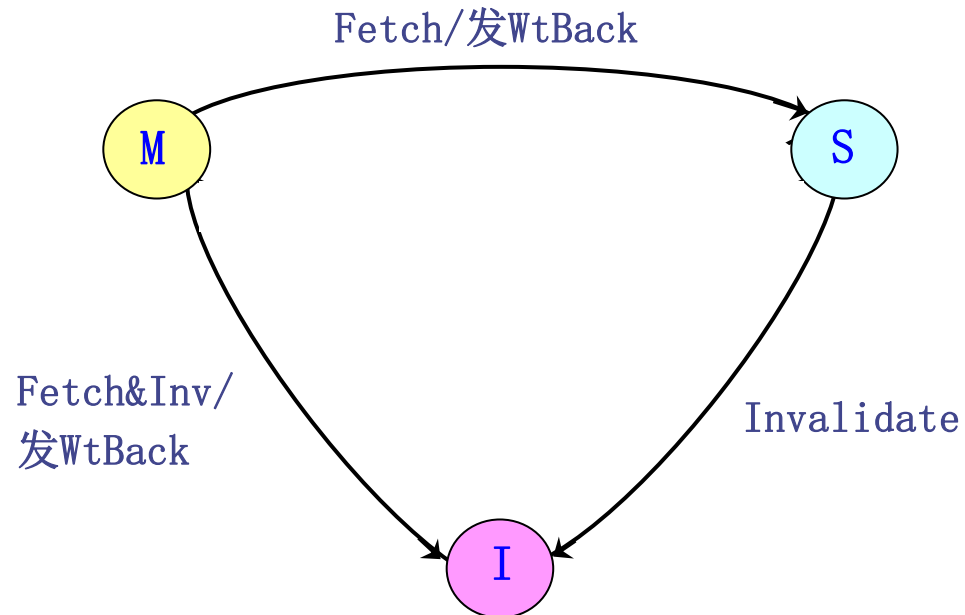
- 在基于目录的协议中，目录承担了一致性协议操作主要功能。
 - 本地结点把请求发给宿主结点中的目录，再由目录控制器有选择地向远程结点发出相应消息。
 - 发出的消息会产生**两种不同类型动作**：
 - 更新目录状态
 - 使远程结点完成相应操作

1. 在基于目录协议的系统中，Cache块的状态转换图。

➤ 响应本地 CPU 请求



- 远程结点中Cache块响应来自宿主结点请求的状态转换图



2. 目录的状态转换及相应的操作

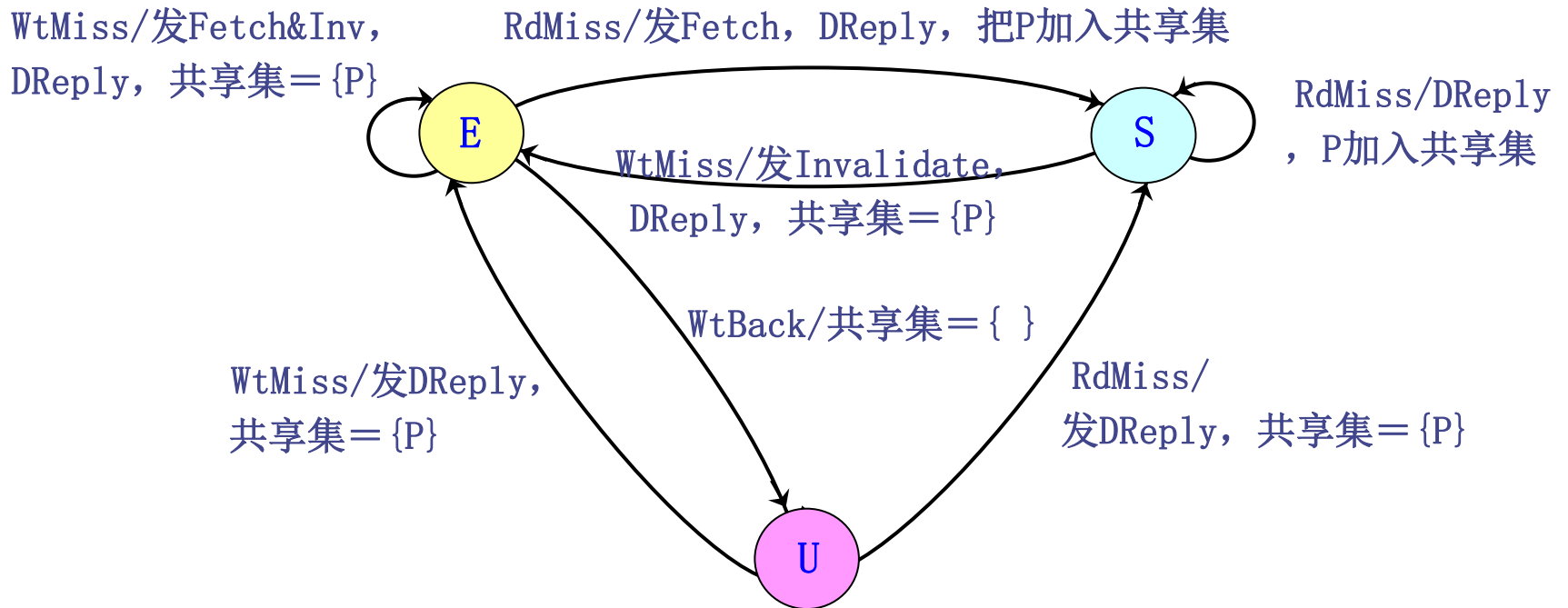
- 目录中存储器块的状态有3种
 - 未缓存
 - 共享
 - 独占
- 位向量记录拥有其副本的处理器集合。该集合称为共享集合。
- 对于从本地结点发来的请求，目录所进行的操作包括：
 - 向远程结点发送消息以完成相应的操作。这些远程结点由共享集合指出；
 - 修改目录中该块的状态；
 - 更新共享集合。

➤ 目录可能接收到3种不同的请求

- ❑ 读不命中
- ❑ 写不命中
- ❑ 数据写回

（假设这些操作是原子的）

10.3 分布式共享存储器系统结构



U: 未缓存 (Uncached) S: 共享 (Shared): 只读
E: 独占 (Exclusive): 可读写 P: 本地处理器

目录的状态转换及相应的操作

- 当一个块处于未缓存状态时，对该块发出的请求及处理操作为：
 - **RdMiss**（读不命中）
 - 将所要访问的存储器数据送往请求方处理机，且该处理机成为该块的唯一共享结点，本块的状态变成共享。
 - **WtMiss**（写不命中）
 - 将所要访问的存储器数据送往请求方处理机，该块的状态变成独占，表示该块仅存在唯一的副本。其共享集合仅包含该处理机，指出该处理机是其拥有者。

- 当一个块处于**共享状态**时，其在存储器中的数据是当前最新的，对该块发出的请求及其处理操作为：
 - **RdMiss**
 - 将存储器数据送往请求方处理机，并将其加入共享集合。
 - **WtMiss**
 - 将数据送往请求方处理机，对共享集合中所有的处理机发送作废消息，且将共享集合改为仅含有该处理机，该块的状态变为独占。

- 当某块处于**独占状态**时，该块的最新值保存在共享集合所指出的唯一处理机（拥有者）中。

有三种可能的请求：

- **RdMiss**

- 将“取数据”的消息发往拥有者处理机，将它所返回给宿主结点的数据写入存储器，并进而把该数据送回请求方处理机，将请求方处理机加入共享集合。
- 此时共享集合中仍保留原拥有者处理机（因为它仍有一个可读的副本）。
- 将该块的状态变为共享。

□ WtMiss

- 该块将有一个新的拥有者。
- 给旧的拥有者处理机发送消息，要求它将数据块送回宿主结点写入存储器，然后再从该结点送给请求方处理机。
- 同时还要把旧拥有者处理机中的该块作废。把请求处理机加入共享者集合，使之成为新的拥有者。
- 该块的状态仍旧是独占。

□ WtBack（写回）

- 当一个块的拥有者处理机要从其Cache中把该块替换出去时，必须将该块写回其宿主结点的

存储器中，从而使存储器中相应的块中存放的数据是最新的（宿主结点实际上成为拥有者）；

- 该块的状态变成未缓冲，其共享集合为空。

10.3.3 目录的三种结构

- 不同目录协议的主要区别主要有两个
 - 所设置的存储器块的状态及其个数不同
 - 目录的结构
- 目录协议分为3类
 - 全映象目录、有限映象目录、链式目录

10.4 同 步

重要性：

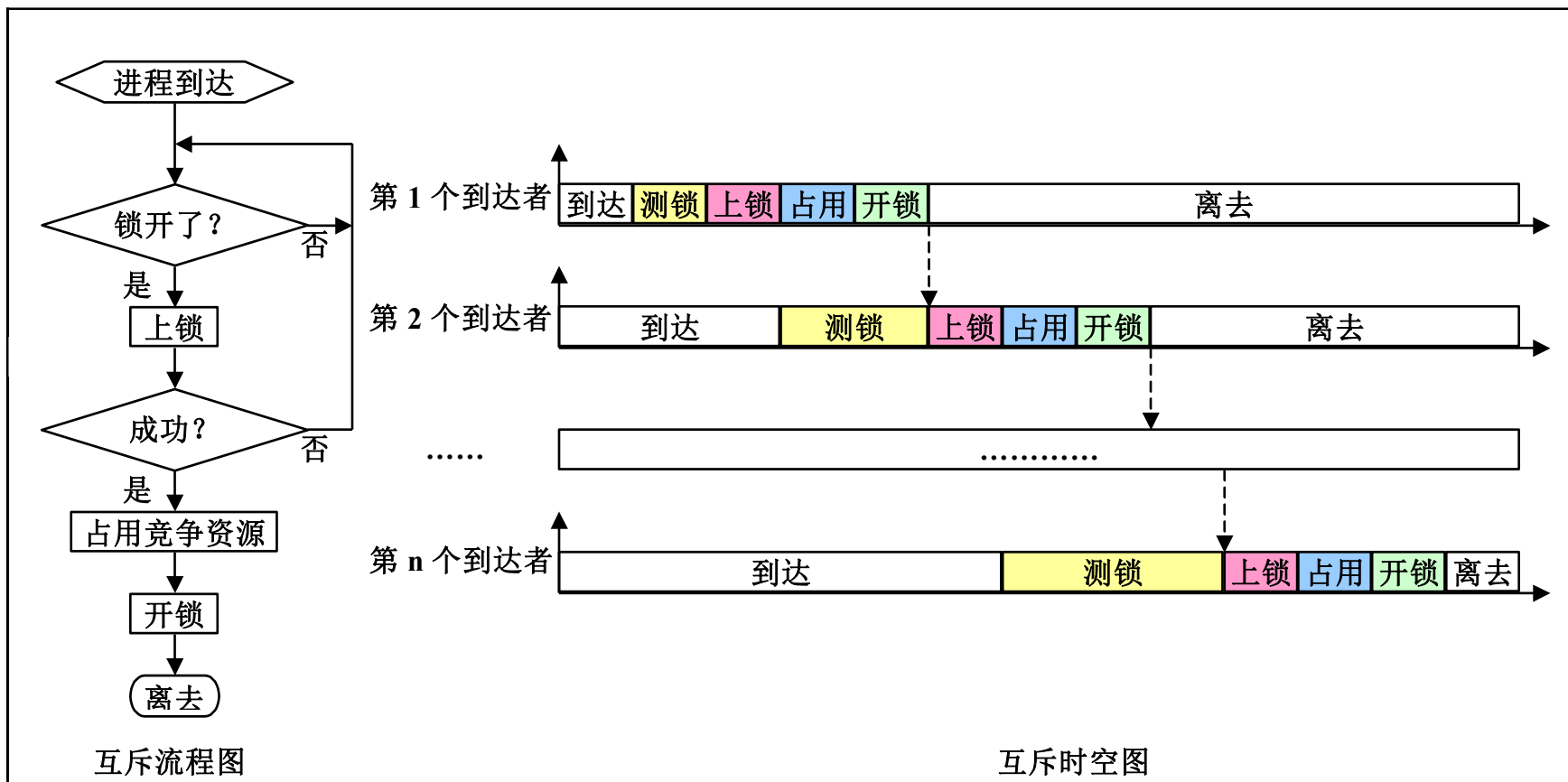
单处理机的多进程（或多线程）并发只是宏观的，由于处理机的唯一性，不可能“精确地”在某一时间做同一事情。因为CPU时间片切换的不可预测，需防止某一进程（或线程）正在处理、尚未完成的共享资源被切换过去的其它进程（或线程）半途操作，造成双方都不期望结果（如共享打印机）。所以“信号灯”或“锁”的作用是告诉其它进程（或线程）：“此资源尚在处理中”。这种“信号灯”或者“锁”的实现可完全用软件手段。

多处理机的多进程（或多线程）并发则是“真实的”，完全可能“精确地”在某一时刻开始做同一事情，就像多台外设同时申请中断一样。对此，须有专门硬件仲裁与排队同时到达的多个申请，用硬件信号“暂停”未竞争到使用权的处理机。

下面分别讨论用**硬件原语实现互斥、同步的基本方法**。“原语”意思是该操作具有“原子性”（atomicity），中途不允许其它进程（或线程）插足。原语通常是操作系统提供的一个调用函数。

互斥过程

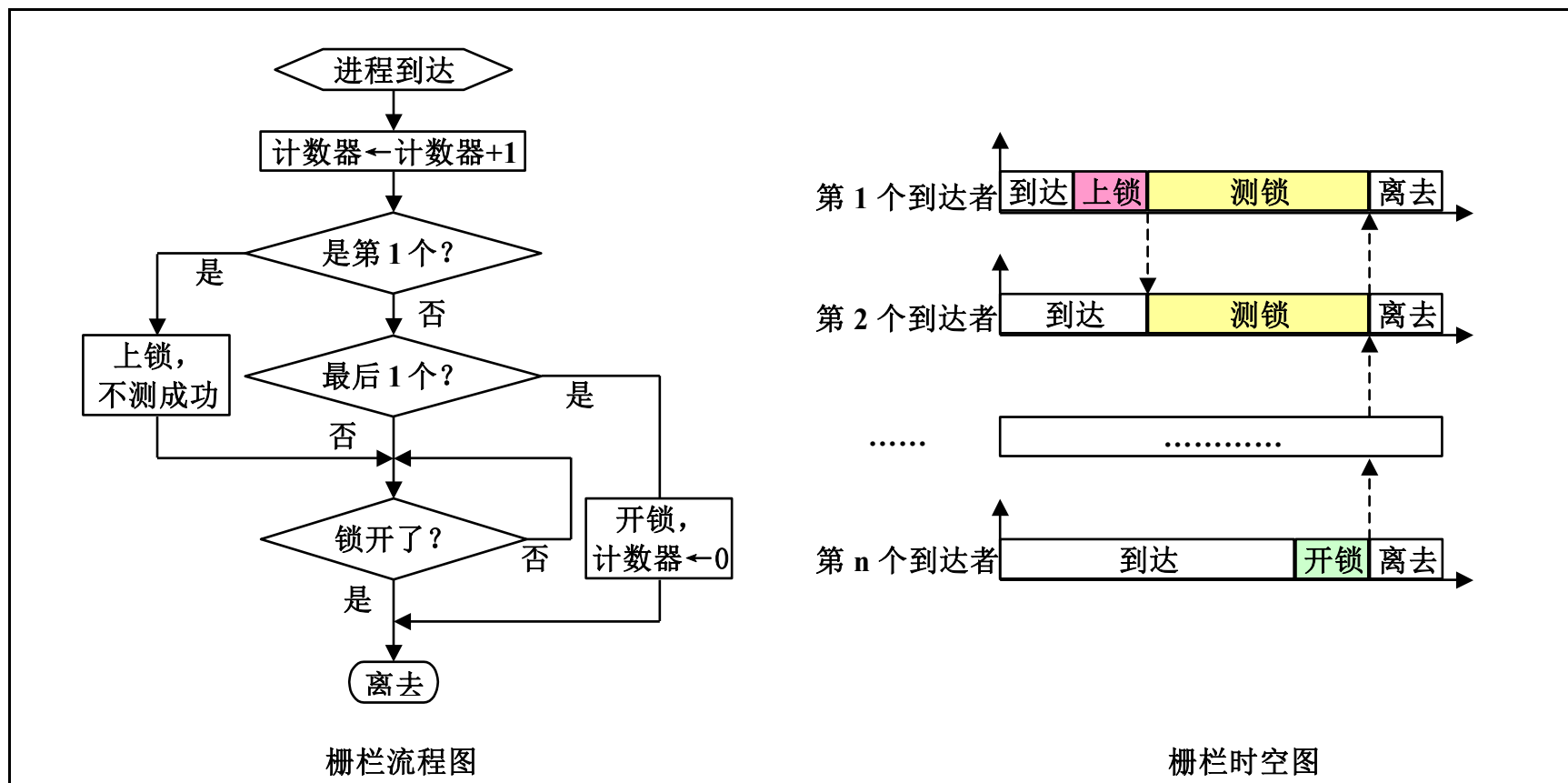
典型的互斥是临界资源的串行使用。



特点：每个进程都需要经历测锁→上锁→开锁的过程，共有n次竞争上锁、n次开锁。只需1个锁变量。

同步过程

典型的同步操作是栅栏（barrier）：栅栏强制所有先到达该栅栏的进程进行等待，直至最后1个进程到达，由它释放全部进程，继续执行。



特点：只需1次上锁、1次开锁， $n-1$ 个进程需要测锁等待。需2个锁变量。

锁的类型及其操作

[1] 二值锁（布尔锁）

状态：开放、锁闭

操作：读锁、加锁、开锁

用途：互斥，保证临界资源操作的唯一性

[2] 多值锁（算术锁）

状态：负（拒绝）、零（空闲）、正（累积数）

负（欠缺数）、零（空闲）、正（富余数）

操作：读锁、增值、减值、清零

用途：同步，记录等待中的进程数，或已通过了的进程数

10.4.0 多处理机的锁存储模型

多处理机系统都用多个局部Cache满足各处理机并行访存的需要，但对进程控制用的信号灯（或称锁）来说，多复本破坏了它们的唯一性，造成同一变量在不同处理机看来有不同值的后果。

因此对信号灯（锁）的管理需遵循以下原则：

- ①读信号灯在当地Cache进行，只要复件有效；
- ②写信号灯必须直接作用于主存原件，采用“写直达”策略；
- ③写过程有排它性，硬件推迟其它处理机的写请求；
- ④写完原件要通知所有其它Cache中的复件作废，即“一写多废”，让其它处理机随后发生“读失效”或“写失效”；
- ⑤信号灯管理系统的时间开销主要来源于“读失效”和“写失效”，或称“总线事务处理”次数；
- ⑥探讨不同的锁管理机制，目的是在程序正确前提下，减少时间开销。

10.4 同 步

同步机制通常是在硬件提供的同步指令的基础上，通过用户级软件例程来建立的。(Lock & Unlock)

10.4.1 基本硬件原语

在多台处理机中实现同步，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。

1. 典型操作：原子交换EXCH（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寄存器中的1进行交换。
 - 如果返回值为1，别的处理器已经上了锁
 - 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：交换操作的原子性

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)
- SC (store conditional)
- 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。
- 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
- LL则返回该存储单元初始值。

4、LL/SC指令对（一种比较容易实现的原语，可等价替代其它原语）

- **LL**(load linked或load locked)是一条特殊读指令，它读取“锁”单元原值到指定寄存器。注意该指令通常只访问Cache；
- **SC**(store conditional)是一条特殊写指令，它首先再读“锁”单元，如果与**LL**所读相同，则将指定寄存器中的新值送给“锁”单元，并返回值**1**表示成功。如果内容不同，直接返回**0**表示失败。
- 语法：见下例。

例A：用LL/SC指令实现原子交换R4与R1所致单元值的交换。

```
try: OR  R3, R4, R0    ; R3←R4（R4是要换入锁的值）  
      LL  R2, 0(R1)    ; R2←0(R1)（存储单元中的锁值）  
      SC  R3, 0(R1)    ; 若写之前0(R1)的值未变，则写入成功，返回1  
      BEQZ R3, try     ; 返回0表示写失败，再试  
      MOV R4, R2       ; 从锁中换出的值交给R4（应写成OR R4, R2, R0）
```

- LL/SC机制的一个优点：用来构造别的同步原语

例B：用LL/SC指令实现 `fetch_and_increment`

```
try: LL R2, 0(R1)          ; R2←0(R1)
      DADDIU R2, R2, #1 ; R2←R2 +1
      SC R2, 0(R1)          ; 0(R1)←R2
      BEQZ R2, try          ; 如不成功，再试
```

10.4.2 用一致性实现锁

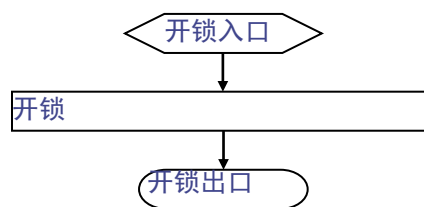
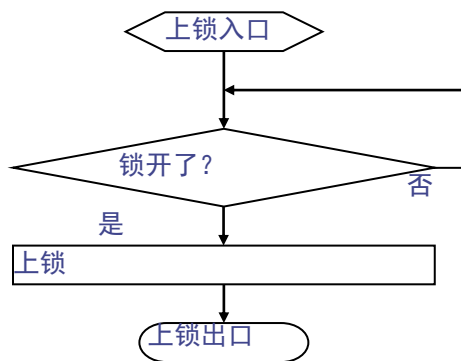
- 采用多处理机的一致性机制来实现旋转锁。
- 旋转锁

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。

旋转锁（又称自旋锁，Spin Lock）

定义：该系统函数循环读取、判断指定的锁变量，直至发现锁被释放，然后上锁，返回用户程序。

释放锁的时候，只需简单地将锁置为0。



语法：spin (slock=0)

锁值：教材P320倒12行用置0来释放锁，而linux内核规定slock=1代表锁被释放，0或负值代表锁被占用。《题解》P234倒8行置1来释放锁，显然又用了linux内核规定。为便于看书，我们先遵守教材规定。

功能：等同于原语“测试并置定”

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中存放该旋转锁地址。

DADDIU R2, R0, #1 ; R2←1

lockit: EXCH R2, 0(R1) ; R2 $\xleftrightarrow{\text{交换}}$ 0(R1)

BNEZ R2, lockit ; 如R2≠0，再试

2. 机器支持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）

- 改进旋转锁（获得第一条好处，避免写操作）
 - 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。
 - 修改旋转锁程序

```
lockit: LD R2, 0(R1)      ; R2←0(R1)
        BNEZ R2, lockit   ; 如R2≠0，再读
        DADDIU R2, R0, #1 ; R2←1
        EXCH R2, 0(R1)    ; R2  $\xleftrightarrow{\text{交换}}$  0(R1)
        BNEZ R2, lockit   ; 如R2≠0，再试
```

每次EXCH都要写主存，而LD指令只访问Cache。改进后仅当“锁”=0时再使用EXCH，可减少与其它处理器的访存冲突

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， 导致Cache不命中	共享	P1 Cache不命中被 处理
6		执行交换， 导致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

- LL / SC原语另一个优点：读写操作明显分开，LL不产生总线数据传送。这使下面代码与使用经过优化交换的代码具有相同特点：

例：用LL/SC指令实现旋转锁

```
lockit: LL R2, 0(R1)      ; R2←0(R1)
        BNEZ R2, lockit   ; 如R2≠0, 再读
        DADDIU R2, R0, #1 ; R2←1
        SC R2, 0(R1)      ; 0(R1)←R2
        BEQZ R2, lockit   ; 如不成功, 再试
```

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很多的情况。

10.4.3 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。

10.4.3 同步操作的性能问题（P322）

尽管具有使用简单方便、性能好的优点，旋转锁也存在自身的不足：

- (1) 由于旋转锁的无序竞争，可能造成等待时间延长，并且存在“不公平”问题。原因有2。
 - a. 随着处理器个数的不断增加，旋转锁竞争性操作增加了系统负担，从而导致更长的等待时间；
 - b. 一个处理器释放旋转锁时要通知等待中的所有处理器，将Cache副本作废，那么几何距离上邻近它的处理器可能会较先地更新Cache，因而增大获得旋转锁的机率。
- (2) 由于每个申请旋转锁的处理器均在全局变量 `slock` 上忙等待，系统总线将因为处理器间的Cache同步而导致繁重的流量（即因读失效或写失效引起的Cache调入、调出），从而降低了系统整体的性能。

例10.3 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？

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。

例10.3（续1）

解：

此题应根据由LL/SC指令实现的旋转锁工作过程分析，题目要求不清。解法可以从推导任意 n 个处理器的公式入手。

1. 开始时， n 个处理器竞争， n 个Cache中都没锁复本，所以它们的LL操作会导致 n 个读失效，产生 n 个读主存的总线事务，分别将锁变量调入 n 个Cache；
2. 动作最快的1个处理机抢先完成SC操作，它把主存中的锁原件“加锁”了，用了1个写主存的总线事务，随即通知其余 $n-1$ 个Cache复本作废，其余 $n-1$ 个SC操作都发生写失效或者读失效（视作废通知是在SC读之前到达还是读之后到达），需要 $n-1$ 个读主存的总线事务重新调入Cache（“按写分配”策略，因为随后还要多次读Cache），合计 n 个总线事务；
3. “加锁”成功的那个处理机对临界资源使用期间，其余 $n-1$ 个处理器只是循环对各自Cache做LL读，并无总线事务发生。最后成功者需要“开锁”，又用了1个写主存的总线事务，并再次通知其余 $n-1$ 个Cache复本作废。

例10.3（续2）

这样总共有 $2n+1$ 个总线事务；

由于收到通知的 $n-1$ 个Cache复件作废，对应 $n-1$ 个处理器的LL操作立即产生读失效，又重演刚才 n 个处理器竞争的过程，只是参与者下降到 $n-1$ 个，.....。

依此类推，当最后1个处理器将锁释放后，前后发生的总线事务次数共有

$$[2n+1] + [2(n-1)+1] + \dots + [2+1] = \sum_{i=1}^n (2i+1) = n(n+1) + n = n^2 + 2n$$

本题中 $n=10$ ，总线事务数共有120个，每个总线事务花费100个时钟周期，合计需要12000个时钟周期。

- 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。

➤ 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
- 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。

```
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
    }
```

➤ 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。

□ 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。

```
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
    }
```

例10.4（教材2版例7.4） 假设总线上10个处理器同时执行一个栅栏，设每个总线事务需100个时钟周期，忽略Cache块中锁的读、写实际花费的时间，以及栅栏实现中非同步操作的时间，计算10个处理器全部到达栅栏，被释放及离开栅栏所需的总线事务数。设总线完全公平，整个过程需多长时间？

答：下表给出一个处理器通过栅栏发出的事件序列，设第一个获得总线的进程并未拥有锁。

表 10.11 第 i 个处理器通过栅栏产生的事件序列

事件	数量	对应源代码	说明
LL	i	Lock(counterlock);	所有处理器抢锁
SC	i	Lock(counterlock);	所有处理器抢锁
LD	1	count=count+1;	一个成功
LL	i-1	Lock(counterlock);	不成功的处理器再次抢锁
SD	1	count=count+1;	获得专有访问产生的失效
SD	1	unlock(counterlock);	获得锁产生的失效
LD	2	spin(release=local_sense);	读释放：初次和最后写产生的失效

用旋转锁实现栅栏同步8

对第*i*个处理器，总线事务数为3*i*+4，而最后一个到达栅栏的处理器需要的事务数少一个，因此*n*个处理器所需的总线事务数为

$$[3n + 4] + [3(n - 1) + 4] + \dots + [3 + 1] - 1 = \sum_{i=1}^n (3i + 4) - 1 = (3n^2 + 11n) / 2 - 1$$

对于10个处理器来说，共需要204个总线事务，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，大约为互斥问题的2倍。

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。

Homework:

1. 10.6、

10.9 假设总线上有**10**个处理器，同时开始执行简单栅栏同步（不考虑混淆“同步”问题），此时“计数器”**count=0**、“到齐锁”**release=0**（关）。每个总线事务处理（即内存访问）的时间是**100**个时钟周期，忽略对已调入**Cache**中的锁进行读写的时间，以及栅栏实现中非同步操作的时间。计算它们全部通过栅栏所需的总线事务数，以及折算的时钟周期数。要求

- (1)** 栅栏采用旋转锁实现（用**LL/SC**指令对实现旋转锁）；
- (2)** 栅栏采用排队锁实现；
- (3)** 栅栏采用**fetch_and_increment**原语实现。

用读取并加1实现栅栏同步1

原语Fetch_and_increment可以原子地取值并增量，减少栅栏记数时所需的时间，从而减小瓶颈的串行度，改进栅栏的实现。

例10.5（教材2版例7.6）：写出fetch_and_increment栅栏的代码。条件与前面假设相同，并设一次fetch_and_increment操作也需100个时钟周期。计算10个处理器通过栅栏的时间，以及所需的总线周期数。

用读取并加1实现栅栏同步2

解 下面的程序段给出栅栏的代码。对 n 个处理器，这种实现需要进行 n 次 `fetch_and_increment` 操作，访问 `count` 变量的 n 次 `cache` 失效和释放时的 n 次 `Cache` 失效，这样总共需要 $3n$ 个总线事务。对 10 个处理器，总共需要 30 个总线事务或 3000 个时钟周期。这里如同排队锁那样，时间与处理器个数成线性增长。

```
local_sense = ! local_sense;           /*local_sense变反*/
fetch_and_increment(count);             /*原子性更新*/
if(count==total) {                     /*进程全部到达*/
    count=0;                           /*初始化计数器*/
    release=local_sense;               /*释放进程*/
}
else {                                 /*还有进程未到达*/
    spin(release=local_sense);         /*等待信号*/
}
```


从上述例子看出，当处理器个数很多时，花费在锁竞争上的时间开销非常大，而如果不竞争，按事先安排的次序互斥地使用临界资源，10个处理器的加锁、解锁动作仅需要20个总线事务。排队锁的想法就是由此产生的。

排队锁的基本原理是用一个数组记录各个进程申请锁的先后顺序，当一个进程释放锁以后，只从队列中取出等在最前面的下一个进程，避免了重新竞争的大量时间开销与“公平性”问题。

排队锁的具体实现算法有很多种，在不同应用中的名称也不一样。比如在Windows中被称为“排队自旋锁” (Queued Spinlock)，在Linux内核2.6.25版中也叫“排队自旋锁”，英文则是FIFO Ticket Spinlock。

❑ 硬件实现

- ❑ 在基于目录的机器上，通过硬件向量等方式来进行排队和同步控制。
- ❑ 在基于总线的机器中要将锁从一个进程显式地传给另一个进程，软件实现会更好一些。

❑ 排队锁的工作过程

- ❑ 在第一次取锁变量失效时，失效被送入同步控制器。同步控制器可集成在存储控制器中(基于总线的系统)或集成在目录控制器中。
- ❑ 如果锁空闲，将其交给该处理器；如果锁忙，控制器产生一个结点请求记录，并将锁忙的标志返回给处理器，然后该处理器不停地进行检测。
- ❑ 当该锁被释放时，控制器从等待的进程排队中选出一个使用锁，这可以通过更新所选进程Cache中的锁变量来完成。

例10.6（教材2版例7.5） 如果在排队锁的使用中，失效时进行锁更新，求10个处理器完成互斥lock和unlock所需的时间和总线事务数。假设条件与前面例子相同。

解 对 n 个处理器，每个处理器都初始加锁产生1个总线事务，其中1个成功获得锁并在使用后释放锁，第1个处理器将有 $n+1$ 个总线事务。每一个后续的处理器需要2个总线事务：1个获得锁，另1个释放锁。因此总线事务的总数为 $(n+1)+2(n-1)=3n-1$ 。注意这里的总线事务总数随处理器数量成线性增长，而不是前面旋转锁那样成二次方增长。对10个处理器，共需要29个总线事务或2900个时钟周期。

```
lock (counterlock) ;    // 由硬件排队 “上锁”
    count=count+1;      // 已到进程数加1
unlock (counterlock) ;  // 开 “独占锁”
    if (count==total) { // 如果是最后一个进程
        count=0;        // 计数器←0
        release=1;      // 开 “到齐锁”
    }
    else {               // 如果不是最后一个进程
        spin (release=1) ; // 等待 “到齐锁” 打开
    }
```

总线事务（即访问主存）发生情形

原则：

- (1) 写锁操作：每次均“直达”主存，如加锁、开锁、锁加1
- (2) 读锁操作：在Cache更新后有一次读失效——从主存重新调入

实例：

- (1) 原子交换：EXEH
- (2) 测试并置定：test_and_set
- (3) 读并加1：fetch_and_increment
- (4) SC原语：每次均访存
- (5) LL原语：当Cache更新后有一次读失效——从主存重新调入