第8章 多处理机

李峰

fli@sdu.edu.cn

https://funglee.github.io

8.1 多处理机结构

8.1.1 多处理分类

- 多处理机系统由**多个独立的处理机**组成，每个处理机都能够独立执行自己的程序
- 按照处理机之间的连接程度：**紧密耦合**和**松散耦合**多处理机
- 按照是否共享主存储器：**共享存储器**和**分布存储器**多处理机
- 按照处理机类型：**同构型**和**异构型**多处理机
- 按照处理机的个数：**大规模并行处理机MPP**和**对称多处理机SMP**

8.1 多处理机结构

8.1.1 多处理分类

- 按照PE与IOP之间互连方式：
  - **对称型**：每个IOP能够连接到所有PE上
  - **非对称型**：每个IOP只与一个PE连接。
  - **冗余对称型**：一个PE与多个IOP连接。
- 按照存储器的访问方式：
  - **均均存储器**，UMA模型
  - **非均均存储器**，NUMA模型
  - **只有Cache**，COMA模型
- **另外，多向量处理机，机群系统**等也称为多处理机系统。

8.1 多处理机

8.1.1 基本概念

- 属于多指令流、多数据流系统
- 多处理机系统由多个独立的处理机组成，每个处理机都能够独立执行自己的程序。
- 实现更高一级的作业、任务之间的并行
- 结构上：要用多个指令部件分别控制，通过机间互连网络实现通信
- 算法上：不限于向量数组
- 系统管理上：依靠软件手段解决资源分配和管理，任务分配、处理机调度、进程同步和通讯

8.1 多处理机

8.1.2 多处理机与并行处理机的主要差别

	并行处理机	多处理机
结构灵活性	针对向量、数组处理而设计的，有专用性，虽然处理单元数很多，但设置有限的、固定的机间互连通路	实现作业、任务、程序段的并行，适应算法，结构灵活多变，实现复杂的机间互连，避免争用共享的硬件资源
程序并行性	实现操作级并行，并行性存在指令内部	并行性还存在于指令外部，表现于多个任务间的并行
并行任务派生	通过指令来反映数据间是否并行计算，并由指令直接启动多个处理单元并行工作	需要专门的指令或语句指明程序中各程序段的并发关系，并控制并发执行
进程同步	实现指令内部对数据操作的并行	实现指令、任务作业级并行
资源分配和任务调度	处理单元数目固定，利用屏蔽手段，改变数目	处理机数目不固定，复杂

8.1 多处理机

8.1.3 多处理机存在的技术问题

- 硬件结构上如何解决处理机、存储器模块及I/O子系统之间的互连
- 如何最大限度地开发系统的并行性，实现多处理机各级的全面并行；
- 如何分割任务的大小，任务的粒度大小；
- 如何协调好处理机中各并行执行的任务和进程间的同步问题；
- 如何将各个任务分配到一个或多个处理机上，解决好处理机调度、任务调度和资源分配问题，防止死锁；
- 系统发生故障，系统如何重新组织，正常工作

8.1 多处理机

8.1.4 多处理机硬件结构

- 紧耦合与松耦合
- 机间互连形式
- 存储器组织

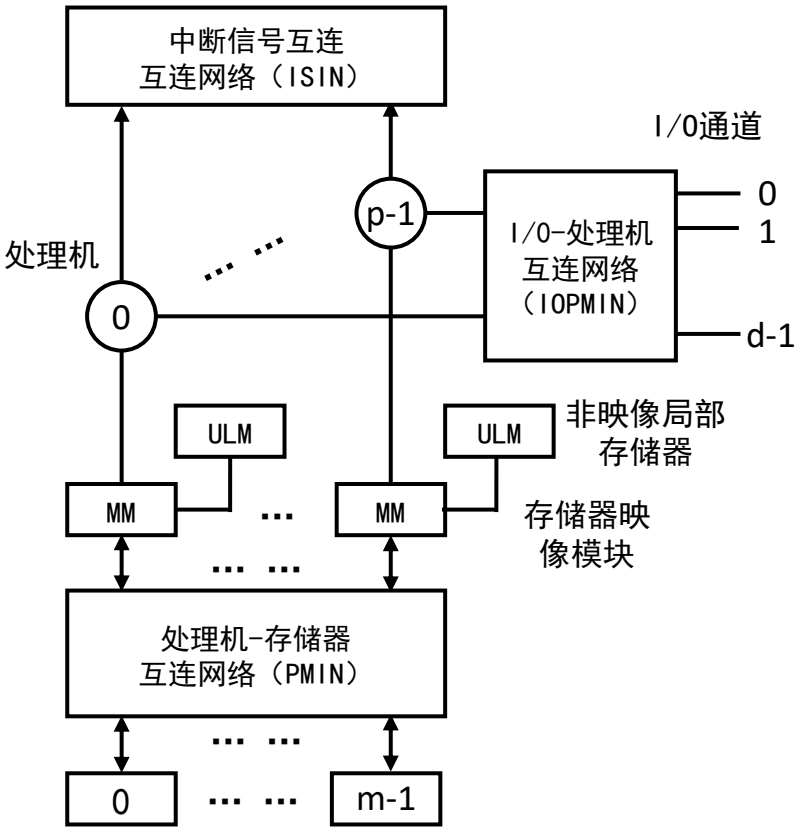
8.1.4 多处理机硬件结构

1 紧耦合多处理机

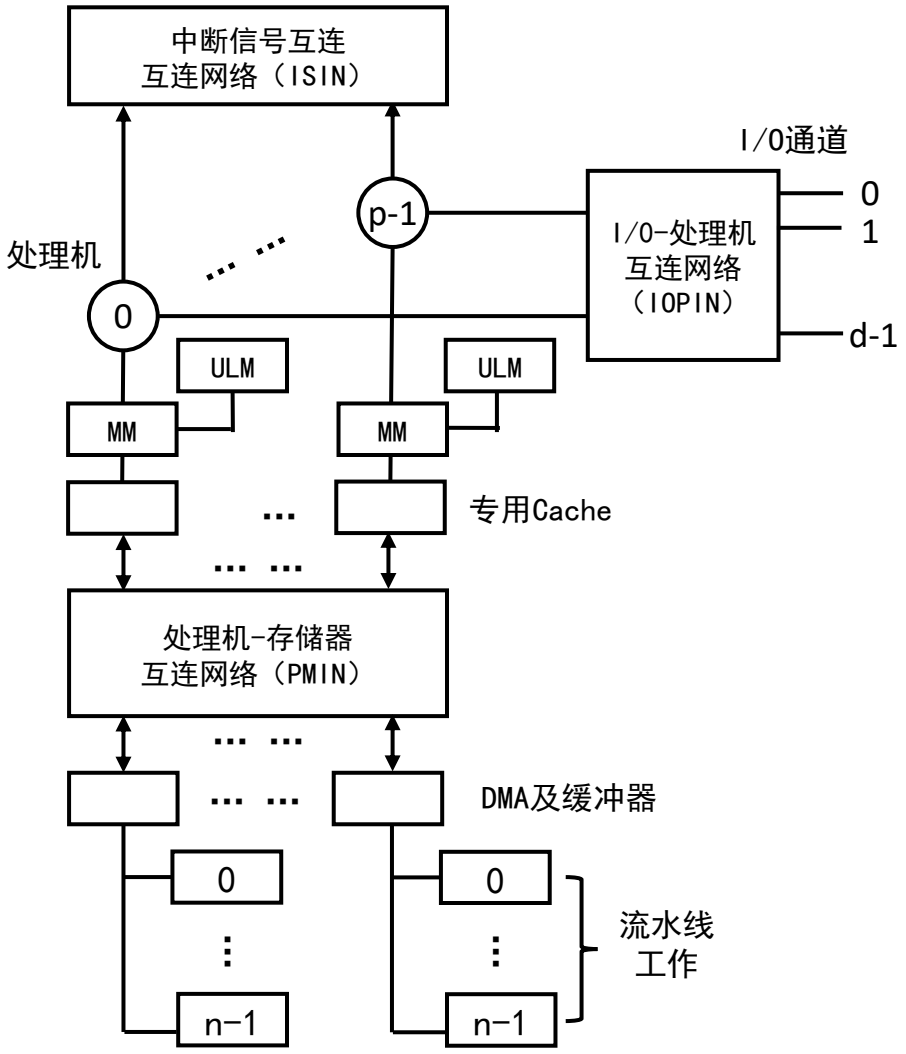
- 是通过**共享主存**来实现处理机间通讯，通信速率受限于主存的频率
- 减少主存冲突，采用**模m多体交叉存取**
- 各个处理机为**同构型**，同一类型、功能相同的多处理机

8.1.4 多处理机硬件结构

1 紧耦合多处理机



处理机不带专用Cache

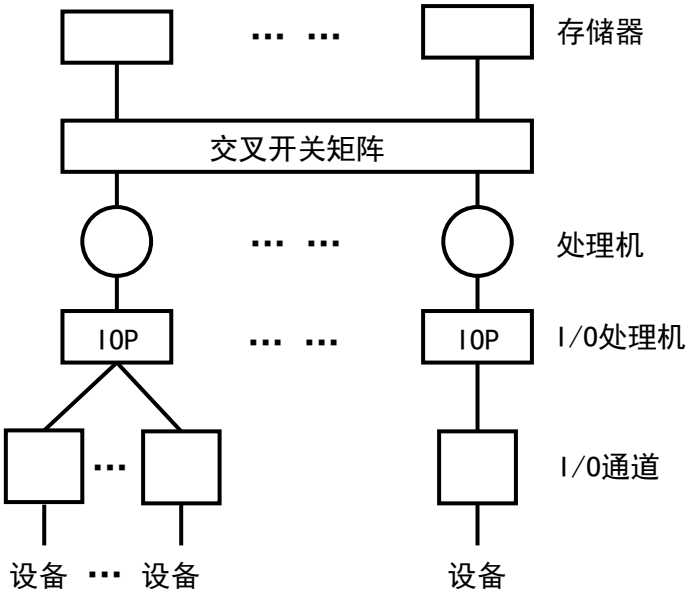


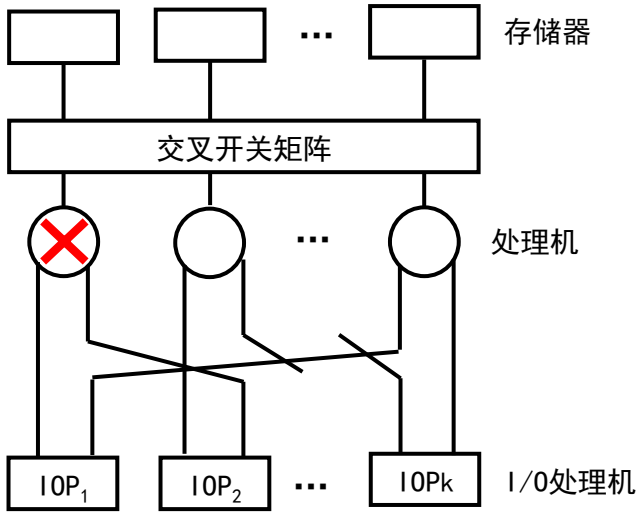
处理机自带专用Cache

8.1.4 多处理机硬件结构

1 紧耦合多处理机

- I/O-处理机互连网络虽然连接灵活，但价格昂贵，所以多数多处理机采用非对称互连





8.1.4 多处理机硬件结构

1 紧耦合多处理机

- 处理机之间共享主存储器，通过高速总线或高速开关连接。
  - 主存储器有多个独立的存储模块
  - 每个CPU能够访问任意一个存储器模块
  - 通过映象部件MAP把全局逻辑地址变换成局部物理地址
  - 通过互连网络寻找合适的路径，并分解访问存储器的冲突
- 多个输入输出处理机IOP也连接在互连网络上，I/O设备与CPU共享主存储器。
- 处理机个数不能太多，几个到十几个

8.1.4 多处理机硬件结构

1 紧耦合多处理机

- 紧密耦合方式要求有很高通信频带。可以采用如下措施：
  - 采用高速互连网络
  - 增加存储器模块个数
  - 每个存储器模块再分成多个小模块，并采用流水线方式工作
  - 每个CPU都有自己的局部存储器LM
  - 每个CPU设置一个Cache

8.1.4 多处理机硬件结构

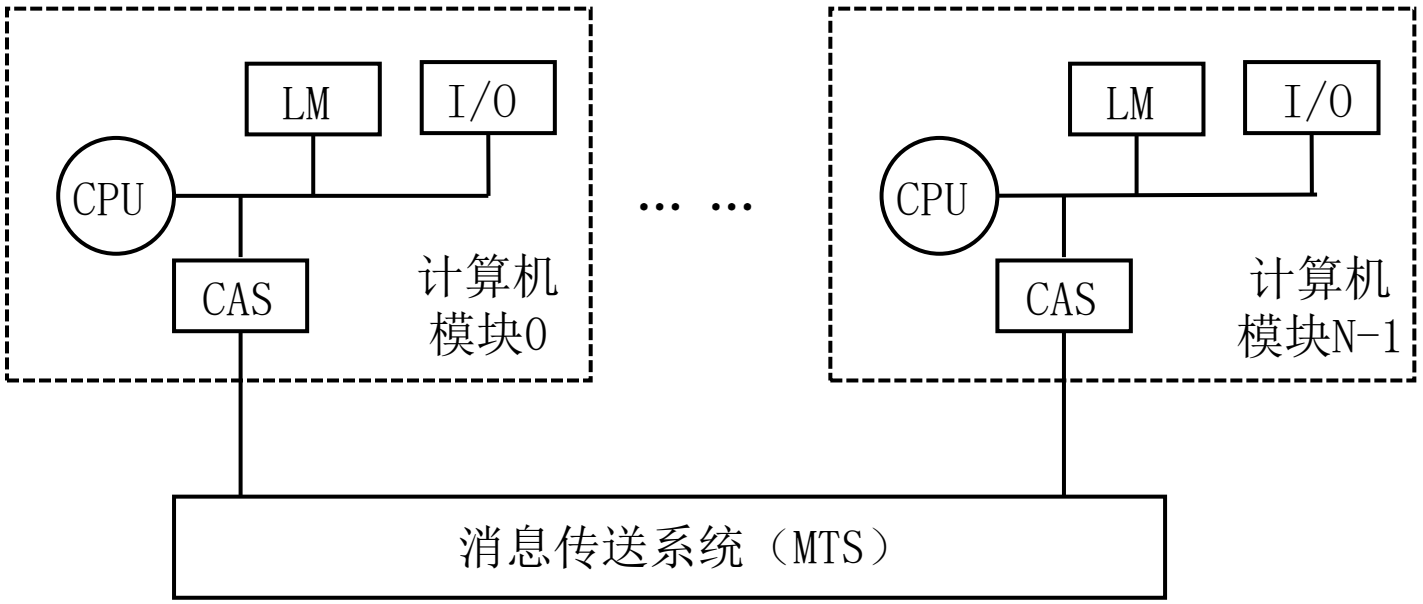
2 松散耦合多处理机

- 每一台处理机都有容量较大的局部存储器，减少访存冲突
- 不同处理机间或者通过**通道互连**实现通信，以共享某些外部设备，或者通过**消息传送系统**（Message Transfer System, MTS）来交换信息（此时各台处理机可带有自己的外部设备）
  - MTS通常采用分时总线或环形、星形、树形等拓扑结构
- 松散耦合多处理机通常用来执行粗粒度的并行计算
  - 处理的作业分割成相对独立的子任务，在各个处理机上执行
- 两种构形：**松耦合非层次型多处理机**、**层次型总线形式的多处理机**

8.1.4 多处理机硬件结构

2 松散耦合多处理机

➢ 松耦合非层次型多处理机

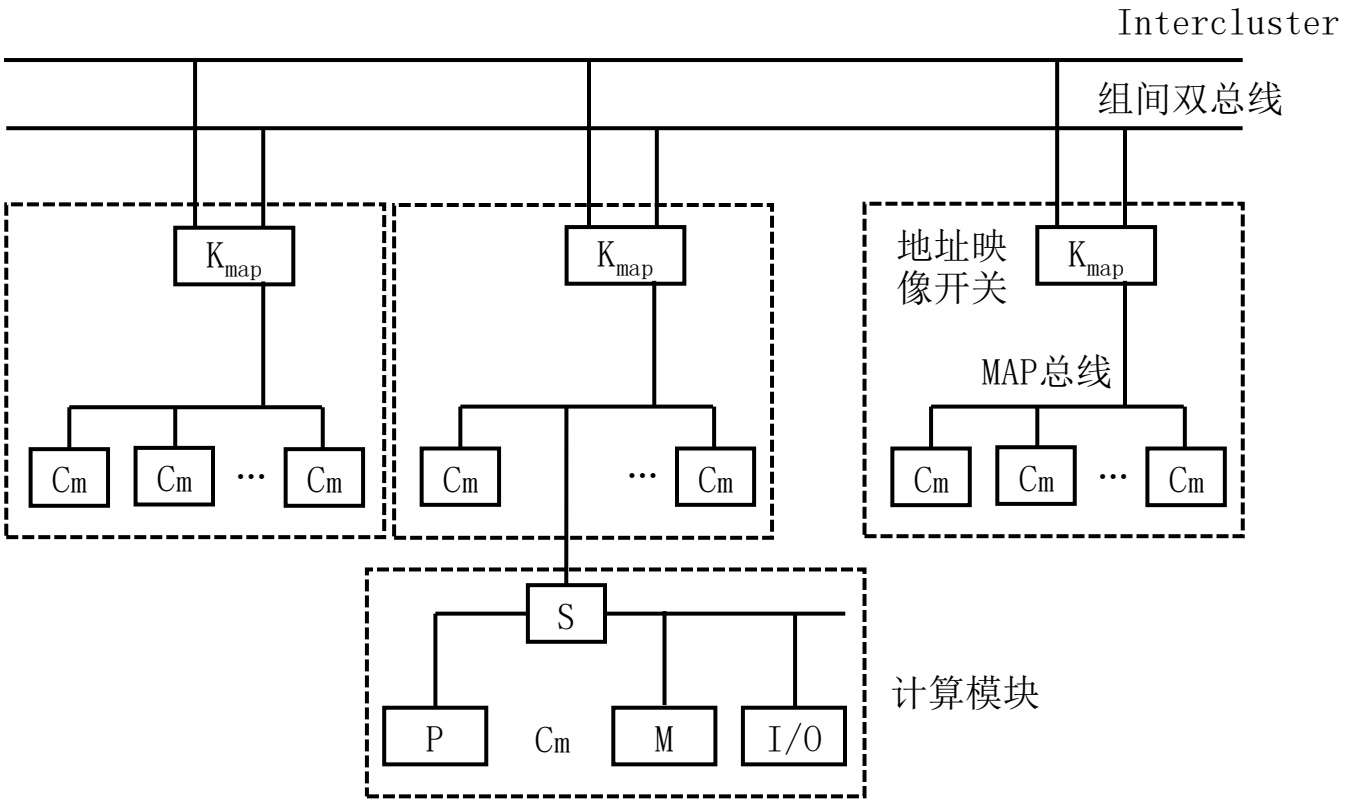


通过MTS连接的松散耦合非层次形多处理机

8.1.4 多处理机硬件结构

2 松散耦合多处理机

➢ 层次型总线形式的多处理机



8.1.4 多处理机硬件结构

2 松散耦合多处理机

- 处理机之间的连接频带比较低
  - 通过输入输出接口连接,处理机间互为外围设备进行连接
  - 例如，IBM公司的机器，都可以通过通道到通道的连接器CTC把两个不同计算机系统的IOP连接起来。
- 通过并口或串口把多台计算机连接起来
  - 例如，用串行口加一个MODEL拨号上网，也可以直接连接；多台计算机之间的连接需要有多个接口。

8.1.4 多处理机硬件结构

2 松散耦合多处理机

- 通过Ethernet网络接口连接多台计算机
- 当通信速度要求更高时，可以通过一个通道和仲裁开关CAS（Channel and Arbiter Switch）直接在存储器总线之间建立连接。CAS中有一个高速的通信缓冲存储器。

8.1.4 多处理机硬件结构

3 机间互连形式

- 总线形式
- 环形互连形式
- 交叉开关形式
- 多端口存储器形式
- 蠕虫穿洞寻径网络
- 开关枢纽结构形式

8.1.4 多处理机硬件结构

3 机间互连形式

➢总线形式

- 多个处理机、存储器模块和外围设备通过接口与公用总线相连，采用分时或多路转接技术传送。
- 结构简单，成本低，增减模块方便，但对总线的失效敏感。
- 提高总线的系统效率：采用优质高频同轴电缆或光纤；采用多总线方式减少冲突概率；

8.1.4 多处理机硬件结构

3 机间互连形式

➢总线形式

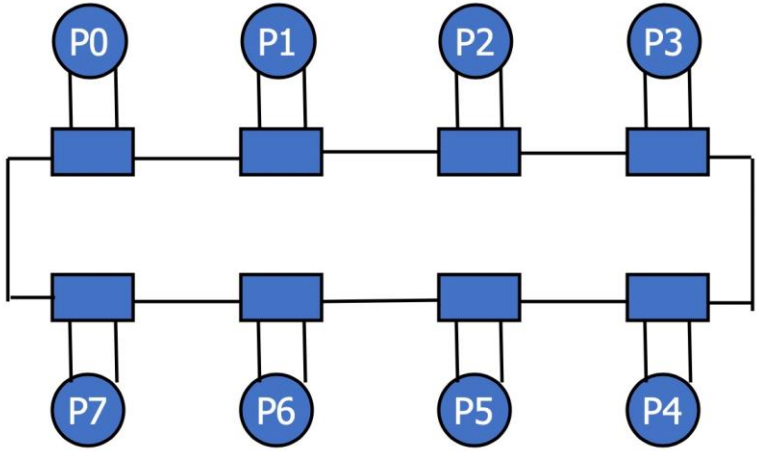
- 总线仲裁算法
  - **静态优先级算法**：为每一个连到总线的部件分配一固定的优先级
  - **固定时间片算法**：把总线按固定大小时间片，轮流提供给部件使用，适合同步总线，时钟同步
  - **动态优先级算法**：让总线上各部件优先级可根据情况按一定规则动态地改变
  - **先来先服务算法**：按接受到访问总线请求先后顺序来响应

8.1.4 多处理机硬件结构

3 机间互连形式

➢ **环形互连形式**

- 总线形成环形互连。
- 令牌（Token）
- 点点连接，物理参数容易控制
- 适合于高带宽的光纤；



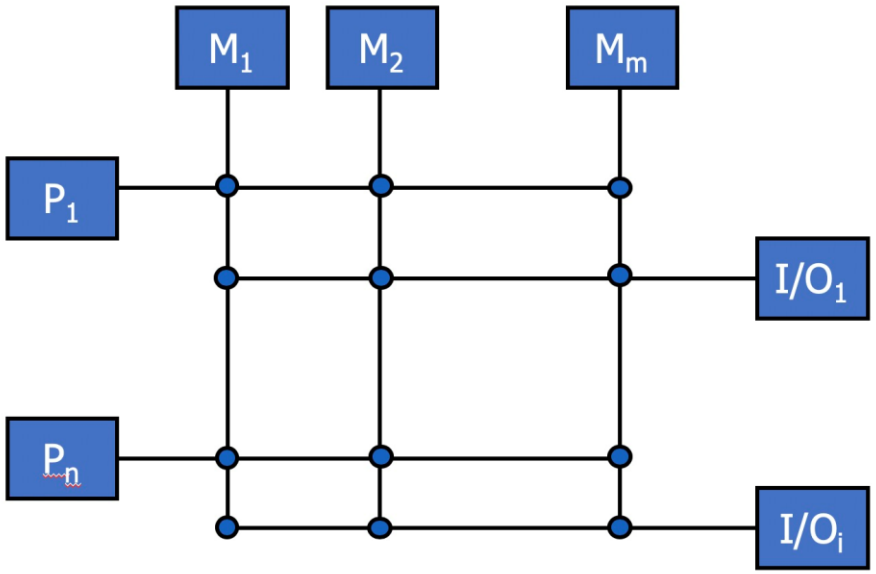
机间采用环形互连的多处理机

8.1.4 多处理机硬件结构

3 机间互连形式

➢交叉开关形式

- 包含一组纵横开关阵列。
- 是总线方式的极端。
- 总线数=m+i+n,m（一般m>=i+n）
- 交叉开关阵列复杂

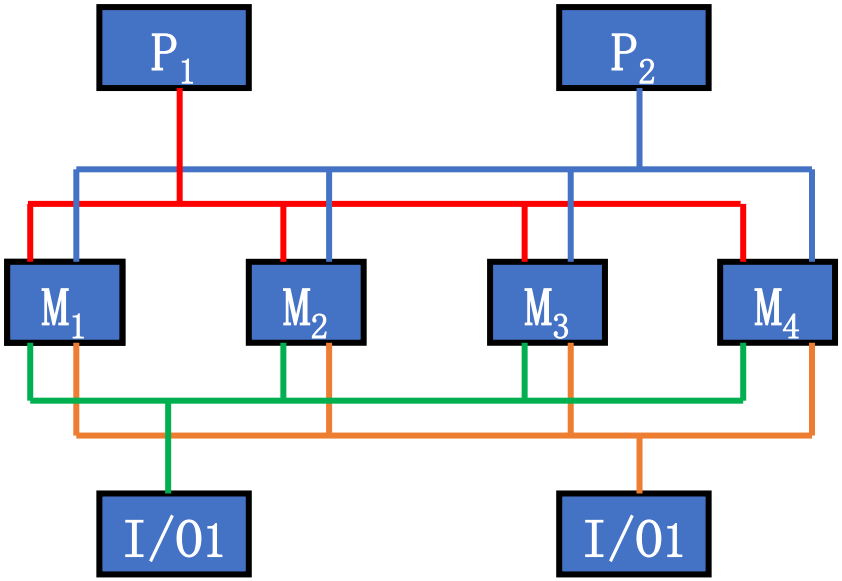


8.1.4 多处理机硬件结构

3 机间互连形式

➢ **多端口存储器形式**

- 如果每个存储器模块有多各访问端口，且将分布在交叉开关矩阵中的控制、转换和优先级仲裁逻辑分别移到相应存储器模块的接口中



8.1.4 多处理机硬件结构

3 机间互连形式

➢ **蠕虫穿洞寻径网络**

- 机间采用小容量缓冲存储器，实现消息分组寻径存储转发之用（曙光1000多处理机）

8.1.4 多处理机硬件结构

3 机间互连形式

➢ **开关枢纽结构形式**

- 把互连结构的开关设置在各个处理机或其接口内部，组成分布式结构
- 美国加州大学伯克利分校设计的树形多处理机X-TREE

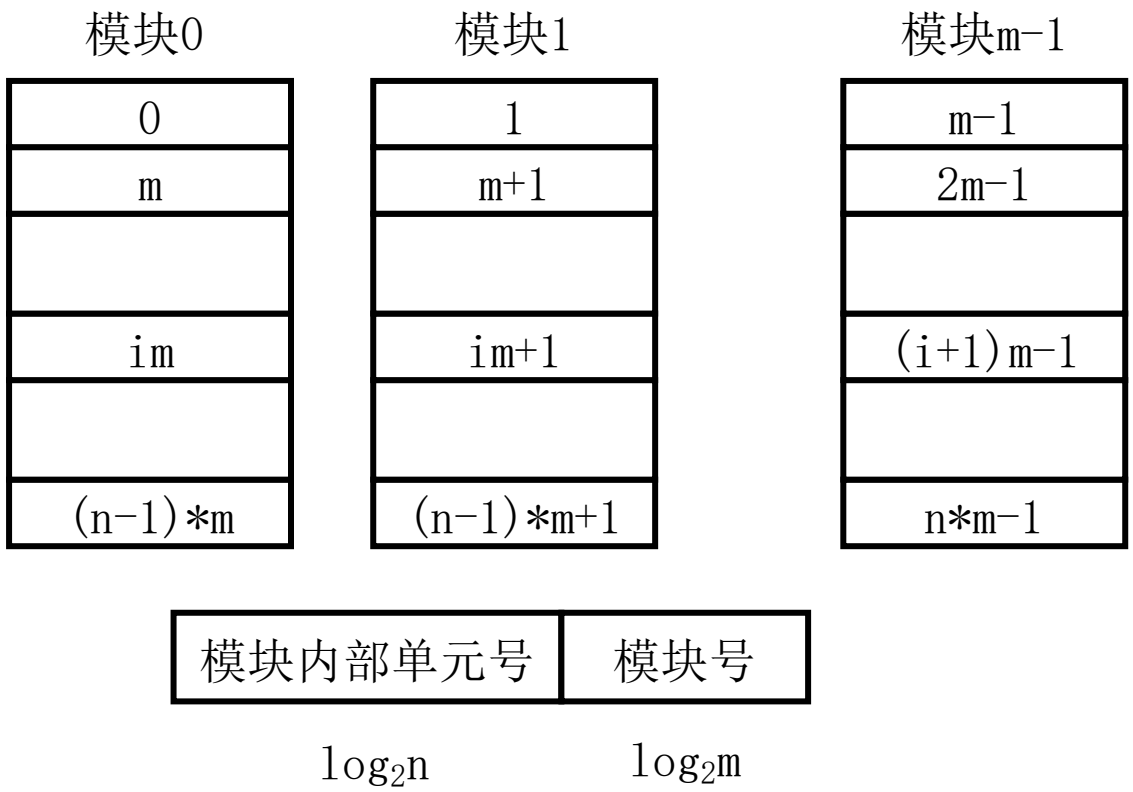
8.1.4 多处理机硬件结构

3 存储器组织

- 并行存储器的构成
  - 有高位交叉和低位交叉
  - 低位交叉：按物理地址顺序轮流地分布在各个存储模块中。
    - 不连续，步距为m
    - 向量、流水或阵列处理机中采用低位交叉
  - 高位交叉：按物理地址顺序从模块0到模块m-1依次连续分布。
    - 连续
    - 多处理机中采用高位交叉

8.1.4 多处理机硬件结构

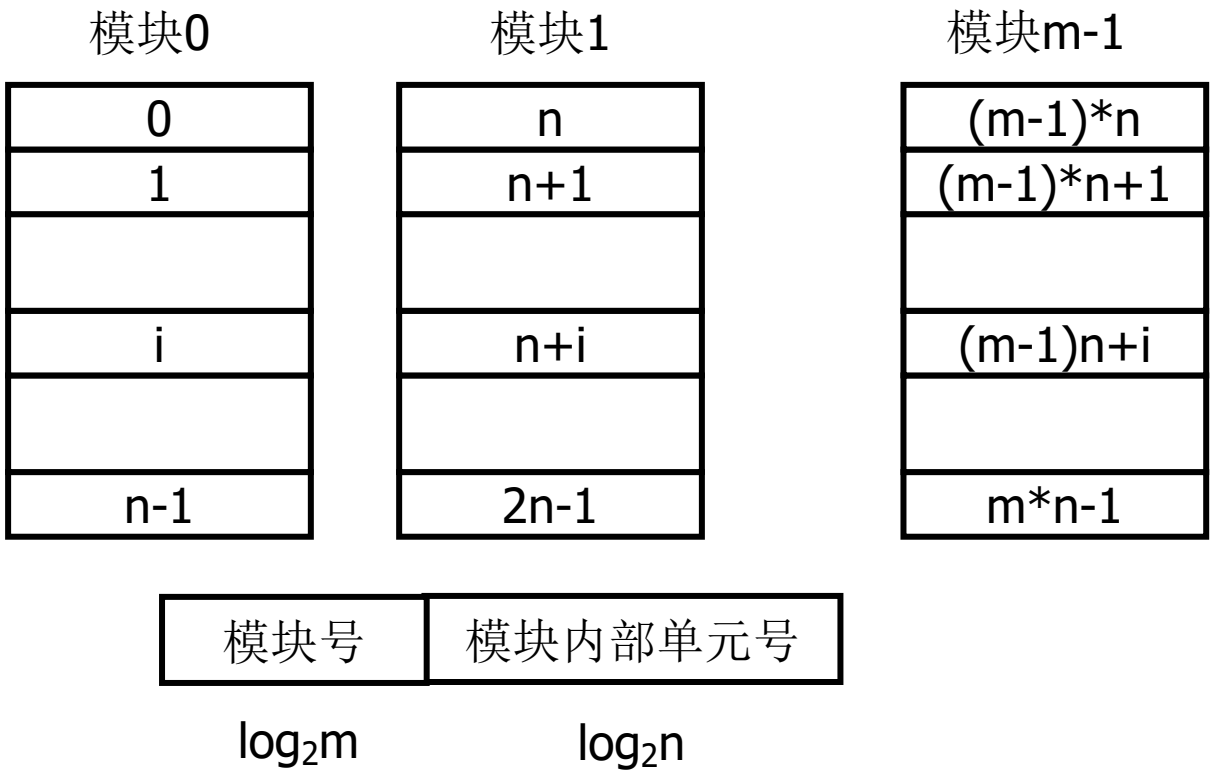
3 存储器组织



m个模块的低位交叉编址

8.1.4 多处理机硬件结构

3 存储器组织



m个模块的高位交叉编址

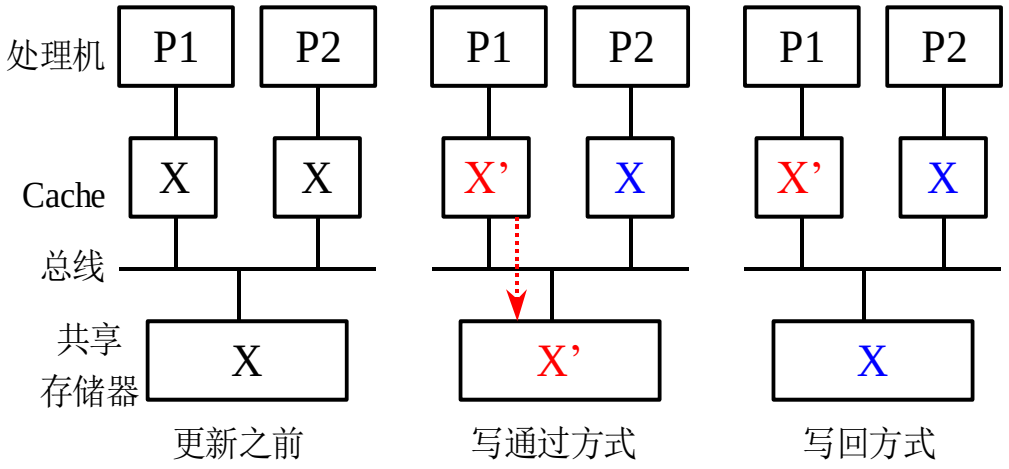
8.2 多处理机的Cache一致性问题

- 在并行处理机和多处理机系统中，采用局部Cache会引起Cache与共享存储器之间的一致性问题。
- 出现不一致性问题的原因有三个：
  - 共享可写的数据
  - 进程迁移
  - I/O传输

8.2.1 多处理机的Cache一致性问题的产生

1 写共享数据引起的不一致性

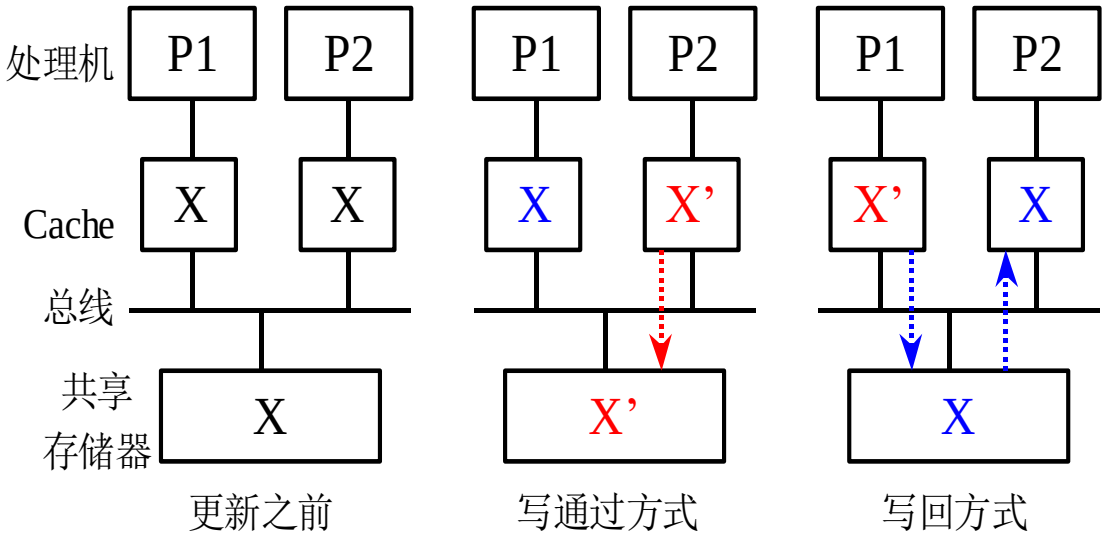
- 使用多个局部Cache时，可能发生Cache不一致性问题：
  - 当P1把X的值写为X'之后，如果P1采用写通过方式，内存中的内容也变为X'，但是P2处理机Cache中的内容还是X。
  - 如果P1采用写回策法，内存中的内容还是X'，当P2处理机要读X时，读到的是X而不是X'。



8.2.1 多处理机的Cache一致性问题的产生

2 进程迁移引起的数据不一致性

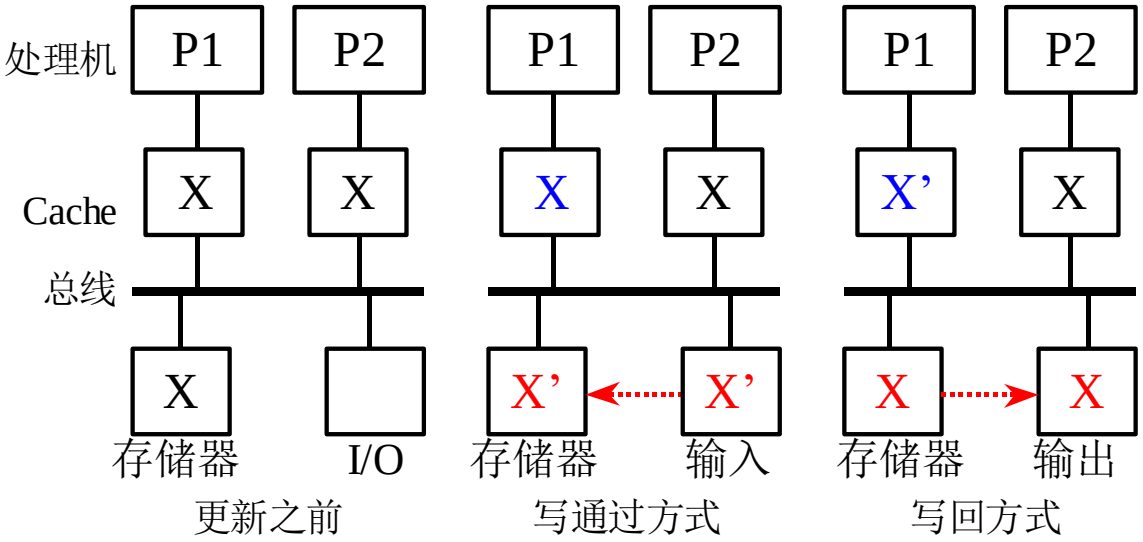
- P1和P2中都有共享数据X的拷贝，P2修改了X，并采用写通过方式，所以内存中的X修改成了X'。如果该进程迁移到P1上，P1的Cache中仍然是X
- P1中有共享数据X的拷贝，而P2中没有该共享数据，P1进程对X进行了修改，如果该进程迁移到了P2上，P2运行时从内存中读到是X



8.2.1 多处理机的Cache一致性问题的产生

3 I/O造成的数据不一致性

- 如果P1和P2在各自的局部Cache中都有X的拷贝，当I/O将一个新数据X'写入存储器时就导致存储器和Cache的数据不一致。
- 如果两个局部Cache中都有X的拷贝，并采用写回方式，当P1把X修改成X'之后；输出部件读X，存储器把X传给输出部件
- 一种解决I/O操作引起数据不一致性的方法是把I/O处理机分别连接到各自的局部Cache上。



8.2.2 监听协议

➢有两类解决Cache不一致性问题的协议：

- 在**总线互连**的多处理机系统中，系统中每个处理机都可以通过总线察觉到存储器系统正在进行的活动，在某个活动破坏了Cache的一致性时，Cache控制器将采取相应的动作使有关拷贝无效或更新，称为**监听协议**。
- 在其它多处理机系统中，处理机无法对存储系统的活动进行监听，通常采用**目录协议**方法。

8.2.2 监听协议

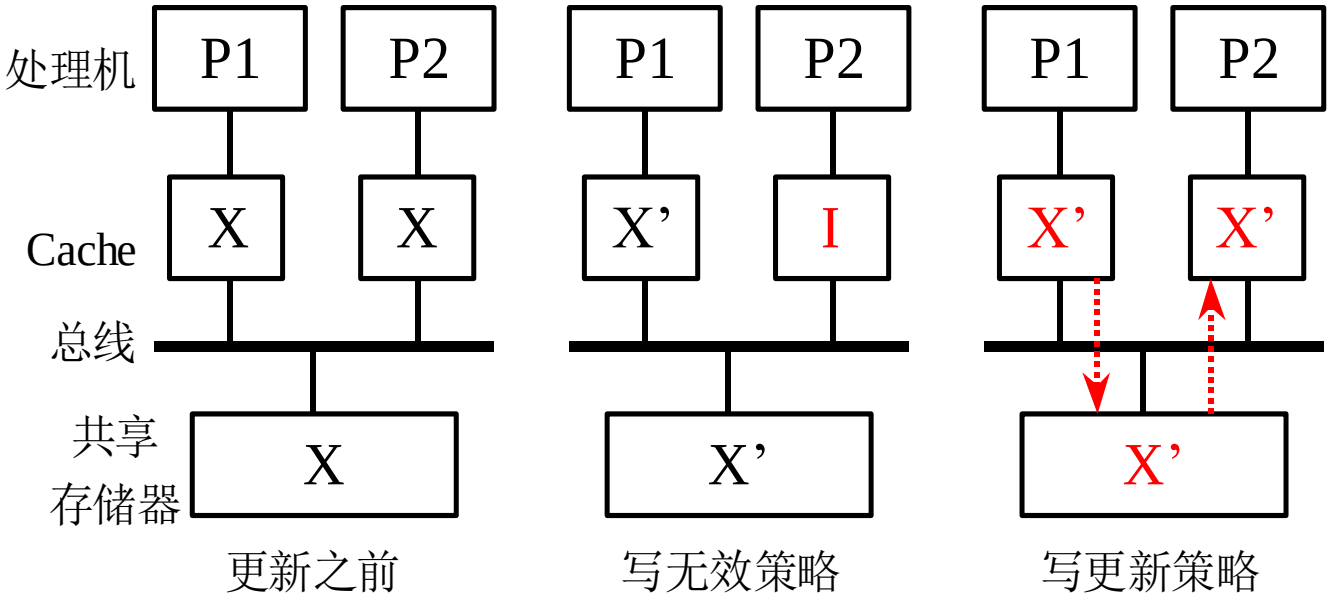
1. 两种监听协议

- 使用监听协议，有两种方法：
  - 方法一：**写无效（Write Invalidate）策略**，在本地Cache的数据块修改时使远程数据块都无效。
  - 方法二：**写更新（Write Update）策略**，在本地Cache数据块修改时通过总线把新的数据块广播给含该块的所有其他Cache
- 采用写无效或写更新策略与Cache采用写回方式（Write Back）还是写通过方式（Write Through）**无关**。
- 如果Cache采用的写通过方式，在使远程数据块无效或更新其他Cache的同时，还要同时修改共享存储器中的内容。

8.2.2 监听协议

1. 两种监听协议

- 由于写更新策略在本地Cache修改时需要通过总线把修改过的数据块广播给所有含该数据块的其他Cache，增加了总线的负担。
- 大部分多处理机系统使用写无效策略。

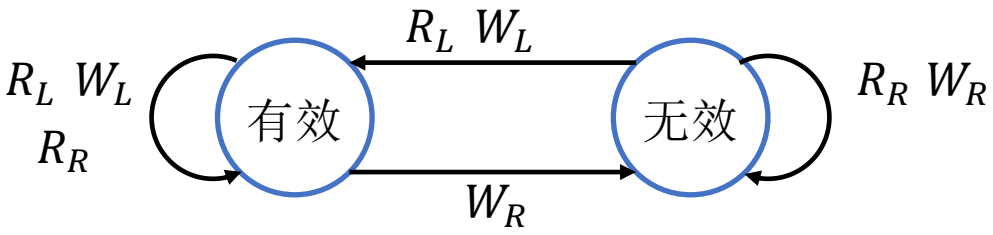


8.2.2 监听协议

2. 采用写通过方式的Cache

➢数据块有两种状态：**有效和无效**。

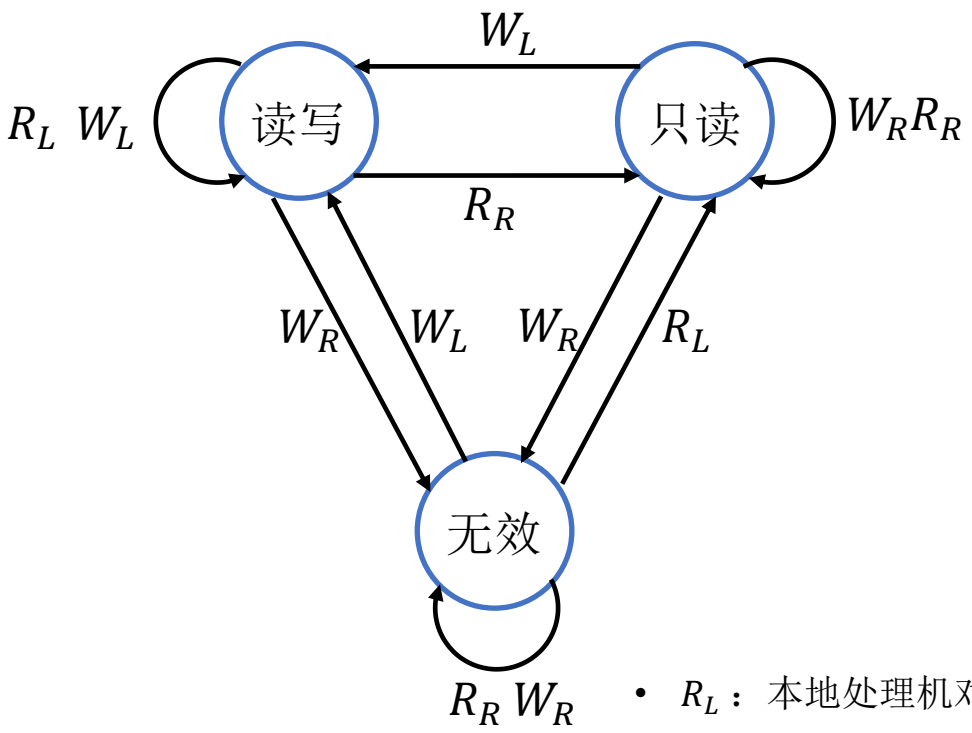
- 有效表示该数据块内容正确，



- R_L 、 W_L表示本地处理机对Cache的读和写操作
- R_R 、 W_R表示远程处理机对Cache中相同内容数据的读和写操作

8.2.2 监听协议

3. 采用写回方式的Cache



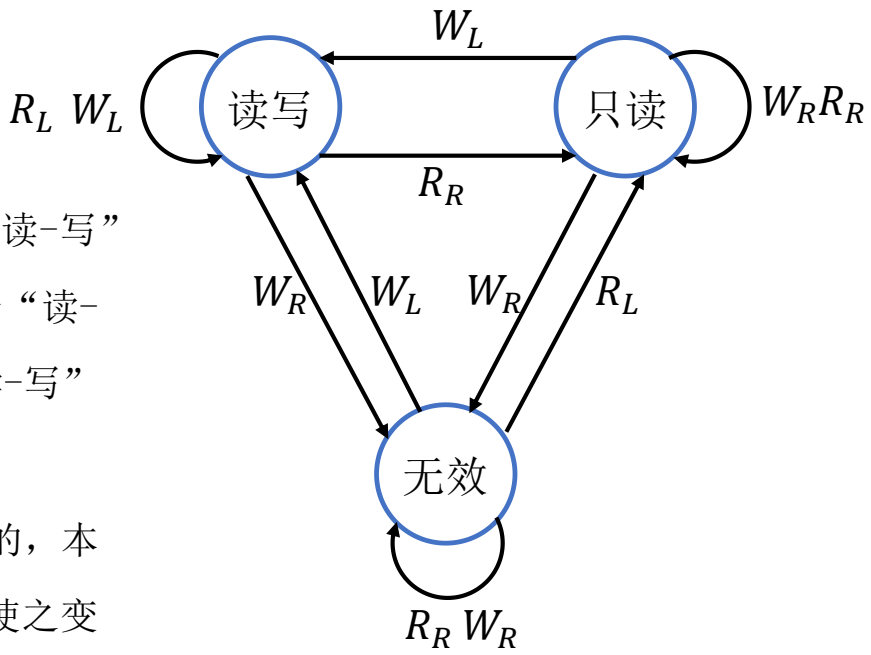
- **只读状态**（read-only）表示整个系统中有多个数据块拷贝是正确的（例如，一个在Cache中，另一个在存储器中）
- **读写状态**(read-write)表示数据块至少被修改过一次，存储器中相应数据块还没有修改，在整个系统中只有一个数据块拷贝是正确的

- R_L：本地处理机对Cache的读操作
- W_L：本地处理机对Cache的写操作
- R_R：远程处理机对Cache中相同内容数据的读操作
- W_R：远程处理机对Cache中相同内容数据的写操作

8.2.2 监听协议

3. 采用写回方式的Cache

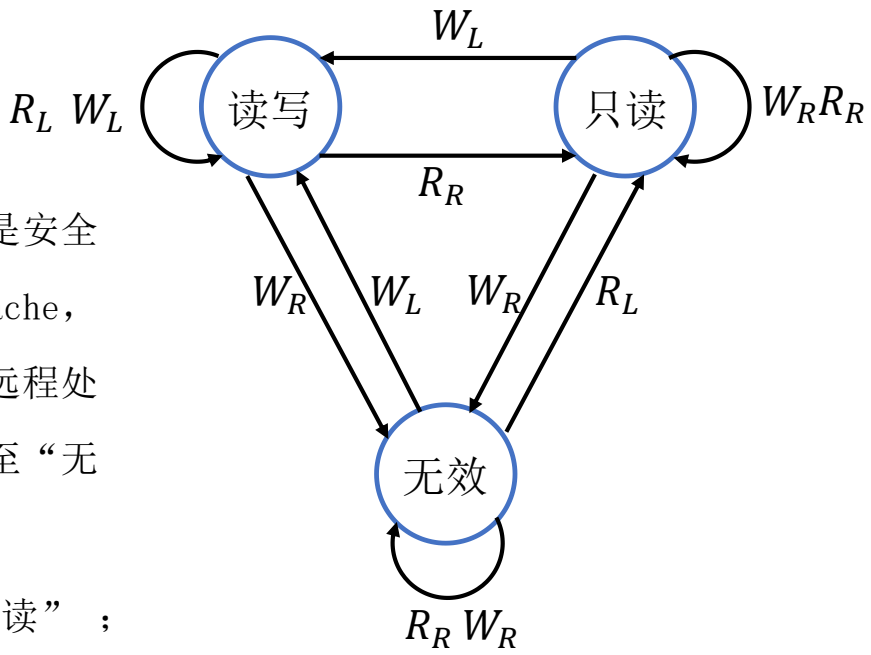
- 系统中不可能同时存在多于一个的存有相同数据的“读-写”状态的数据块，如果一个Cache中的某一个数据块处于“读-写”状态，则其它Cache中就不可能存在有效的（“读-写”或“只读”状态）存存有相同数据的数据块
- 对于只读的数据块，本地的和远程的读操作都是安全的，本地的写操作使状态转移为“读-写”，远程的写操作使之变为“无效”



- 只读状态（read-only）表示整个系统中有多个数据块拷贝是正确的（例如，一个在Cache中，另一个在存储器中）
- 读写状态(read-write)表示数据块至少被修改过一次，存储器中相应数据块还没有修改，在整个系统中只有一个数据块拷贝是正确的

8.2.2 监听协议

3. 采用写回方式的Cache

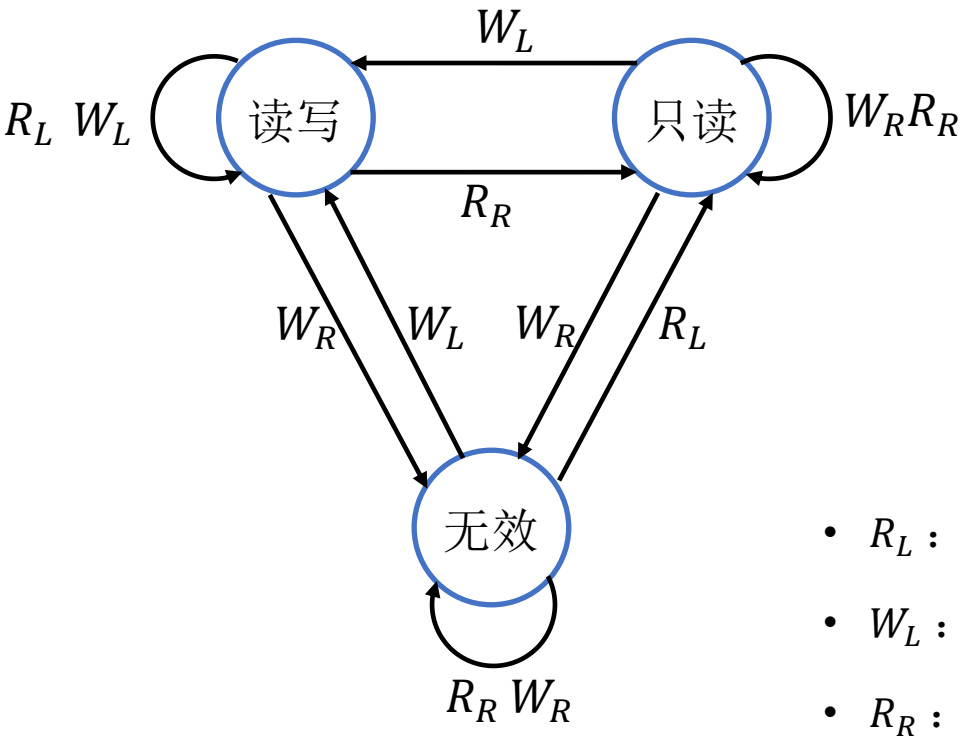


- 对于“读-写”状态的数据块，本地的读、写操作都是安全的，而远程的读操作将数据块传递给远程处理机的Cache，使两个Cache都转移至“只读”状态，远程写操作使远程处理机Cache转移至“读-写”状态，而本地Cache转移至“无效”状态
- 对于“无效”状态，本地读操作，使状态转移至“只读”；本地写操作，使状态转移至“读-写”，同时使其他Cache中相应数据块转移为“无效”状态

- **只读状态**（read-only）表示整个系统中有多个数据块拷贝是正确的（例如，一个在Cache中，另一个在存储器中）
- **读写状态**(read-write)表示数据块至少被修改过一次，存储器中相应数据块还没有修改，在整个系统中只有一个数据块拷贝是正确的

8.2.2 监听协议

3. 采用写回方式的Cache



- R_L：本地处理机对Cache的读操作
- W_L：本地处理机对Cache的写操作
- R_R：远程处理机对Cache中相同内容数据的读操作
- W_R：远程处理机对Cache中相同内容数据的写操作

8.2.2 监听协议

4. 写一次（Write-Once）协议

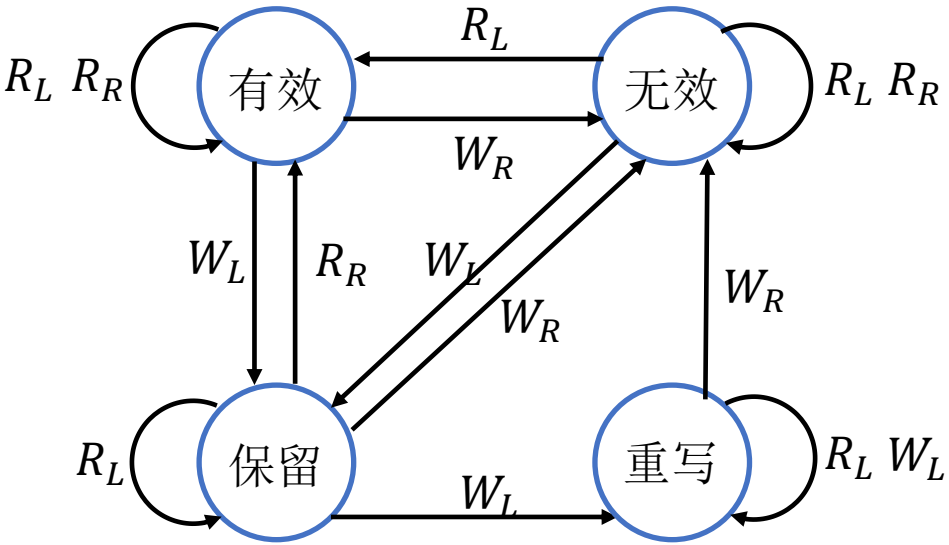
- 方法：**第一次写Cache采用写通过方式，以后采用写回方式。**
- 为了区分第一次写，把“读写”状态分为：保留(Reserved)和重写(Dirty)。共有4种状态

1) **有效**(Valid)：从存储器读入的并与存储器拷贝一致的Cache数据块（相当于写回方式中的只读）
2) **无效**(Invalid)：在Cache中找不到或数据块已作废。
3) **保留**(Reserved)：数据从存储器读入Cache后只被写过一次，Cache和存储器中都正确。
4) **重写**(Dirty)：Cache中的数据块被写过多次，而且是唯一正确的数据块。

8.2.2 监听协议

4. 写一次（Write-Once）协议

- 方法：第一次写Cache采用写通过方式，以后采用写回方式。
- 整个系统中只有一份正确的拷贝。
- 主要优点：减少大量的无效操作，提高了总线效率。
- 主要缺点：当主存储器的内容无效时,读缺失引起的总线读操作必须禁止访问主存储器，而大多数总线不支持这种操作



- R_L：本地处理机对Cache的读操作
- W_L：本地处理机对Cache的写操作
- R_R：远程处理机对Cache中相同内容数据的读操作
- W_R：远程处理机对Cache中相同内容数据的写操作

8.2.2 监听协议

4. 写一次（Write-Once）协议

➢**CPU读Cache：有两种可能性。**

1) 数据块在Cache中存在(包括有效、保留或重写)，CPU直接读取数据,Cache状态不变。

2) Cache中的数据块处于无效状态，将触发“读缺失（read-miss）”事件，系统将设法将有效的数据块调入Cache。

- 如果存在处于有效、保留或重写状态的相应数据块，则将其调入本地Cache；在相应数据块处于重写状态时，还要同时禁止存储器操作。
- 如果不存在处于有效、保留或重写状态的相应数据块，则直接从存储器中读入(只有存储器中是唯一正确的拷贝)。
- 把读入Cache中的相应数据块置为“有效”状态。

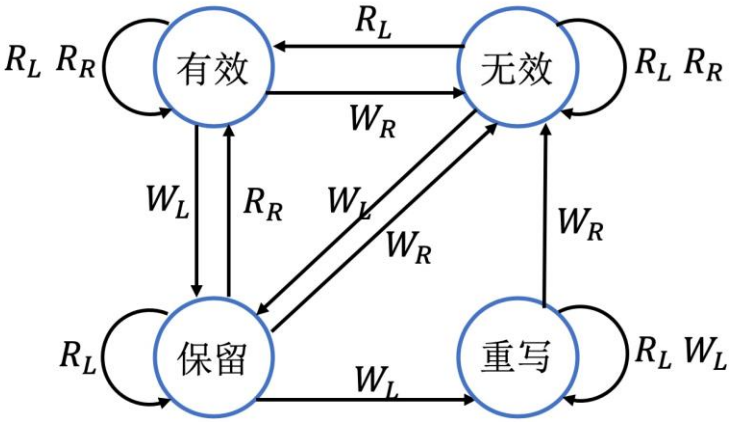
8.2.2 监听协议

4. 写一次（Write-Once）协议

➢CPU写Cache：也有两种可能。

1) 写命中，当Cache处于“有效”状态时，采用写通过方式，把写入Cache的内容同时写入存储器，将Cache的状态转移为“保留”，将其他Cache的相应数据块状态置为“无效”；当Cache处于“保留”或“重写”态时，使用写回方式，Cache的状态转移至“重写”,其他的存有相同内容的Cache处于“无效”态。

2) 写不命中，将数据块调入Cache，采用写通过方式,同时写存储器;将本地Cache的状态置为“保留”，同时将其他Cache的状态置为“无效”。



- R_L：本地处理机对Cache的读操作
- W_L：本地处理机对Cache的写操作
- R_R：远程处理机对Cache中相同内容数据的读操作
- W_R：远程处理机对Cache中相同内容数据的写操作

8.2.3 基于目录的协议

1. Cache目录结构

- 当某台处理机采用Write-invalid协议正在更新一个变量同时其它处理机也试图读该变量的时候，则会发生读缺失并可能导致总线流量大大增加；Write-Update协议更新远程cache中的数据，但这些数据可能永远也不会使用。**这些问题严重限制了采用总线来构造大型多处理机系统。**
- 由于在多级网络上实现广播功能的代价很大，所以监听协议就无法使用，**把使其它Cache数据块无效的一致性命令只发给存放相应数据块的Cache**是一个很好的解决办法。

8.2.3 基于目录的协议

1. Cache目录结构

- Cache目录中存放的内容是大量的指针，用以指明块拷贝的地址，每个目录项还有一个重写位，指明是否有一个Cache允许写入数据。
- 根据Cache目录的存放形式，**有集中式和分布式两种**。
- 根据目录的结构，目录协议分成三类：
  - **全映射(Full-Map)目录**：存放全局存储器每个块的有关数据。
  - **有限(Limited)目录**：每个目录项的指针数固定。
  - **链式(Chained)目录**：把目录分布到所有Cache中。

8.2.3 基于目录的协议

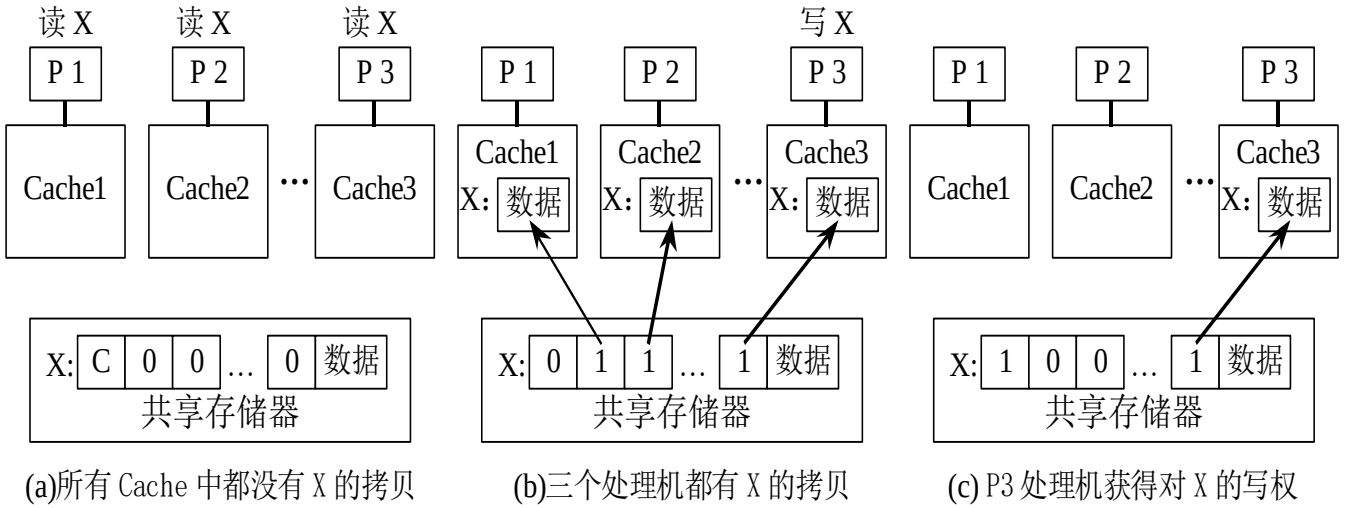
1. Cache目录结构

- Cache目录中存放的内容是大量的指针，用以指明块拷贝的地址，每个目录项还有一个重写位，指明是否有一个Cache允许写入数据。
- 根据Cache目录的存放形式，**有集中式和分布式两种**。
- 根据目录的结构，目录协议分成三类：**全映射(Full-Map)目录、有限(Limited)目录、链式(Chained)目录**
- 目录的使用规则：
  - 当一个CPU对Cache进行写操作时，要根据Cache目录中的内容将所有其他存有相同内容的所有Cache拷贝无效，并置重写位。
  - 在CPU对Cache进行读操作时，如果读命中，则直接读Cache即可。
  - 如果重写位为“0”，则从主存或其他Cache中读入该块，并修改目录。

8.2.3 基于目录的协议

2. 全映射目录

- 目录项中有N个处理机位和一个重写位。
- 处理机位表示相应处理机对应的Cache块的状态（存在或不存在）。
- 如果重写位为“1”而且有且只有一个处理机的重写位为“1”，则该处理机可以对该块进行写操作。
- Cache的每个数据块有两个状态位：一位表示数据块是否有效，另一位表示有效块是否允许写。



(a)所有 Cache 中都没有 X 的拷贝　(b)三个处理机都有 X 的拷贝　(c) P3 处理机获得对 X 的写权

8.2.3 基于目录的协议

2. 全映射目录

- 从第二种状态(b)转移至第三种状态(c)的过程如下：

  1)Cache3发现包含X单元的块有效，但不允许写

  2)Cache3向包含X单元的存储器模块发写请求，并暂停P3工作

  3)该存储器模块发无效请求至Cache1和Cache2

  4)Cache1和Cache2接到无效请求后，将对应块置为无效态，并发回答信号给存储器模块

  5)存储器模块接到Cache1和Cache2的回答信号后，置重写位为“1”，清除指向Cache1和Cache2的指针，发允许写信号到Cache3。

  6)Cache3接到允许写信号，更新Cache状态，激活P3。

- 优点：全映射目录协议的效率比较高。

- 缺点：开销与处理机数目的平方成正比，不具有扩展性。

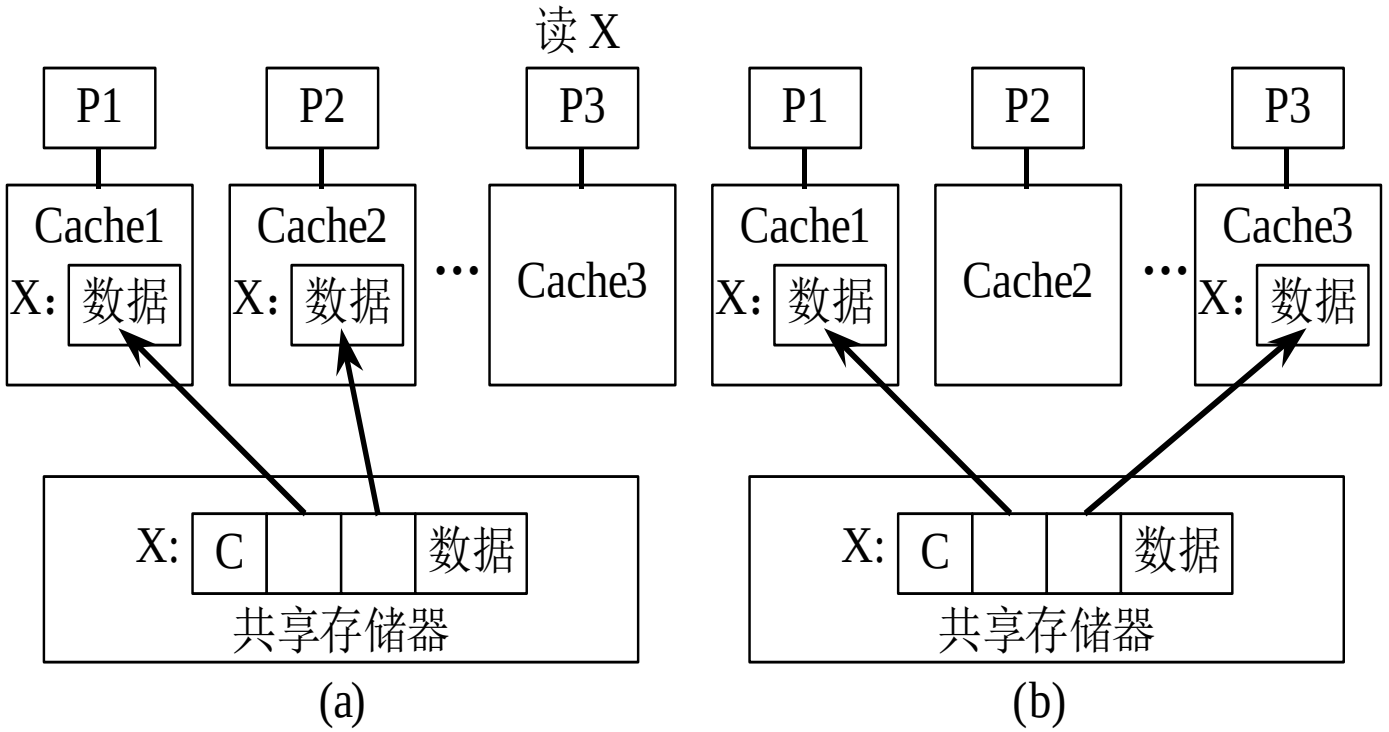
8.2.3 基于目录的协议

3. 有限目录

- 当处理机数目为N时，限制目录大小为O(N $\log_2$ N)。
- 目录指针需要对N进行二进制编码，每个指针占$\log_2$ N位，目录所占的总存储空间与（N$\log_2$ N）成正比。
- 当Cache1和Cache2中都有X的拷贝时，若P3请求访问X，则必须在在Cache1和Cache2中选择一个使之无效，这种替换过程称为驱逐。
- 有限目录的驱逐需要一种驱逐策略，驱逐策略的好坏对系统的性能具有很大的影响。驱逐策略与Cache替换策略在很多方面是相同的。

8.2.3 基于目录的协议

3. 有限目录

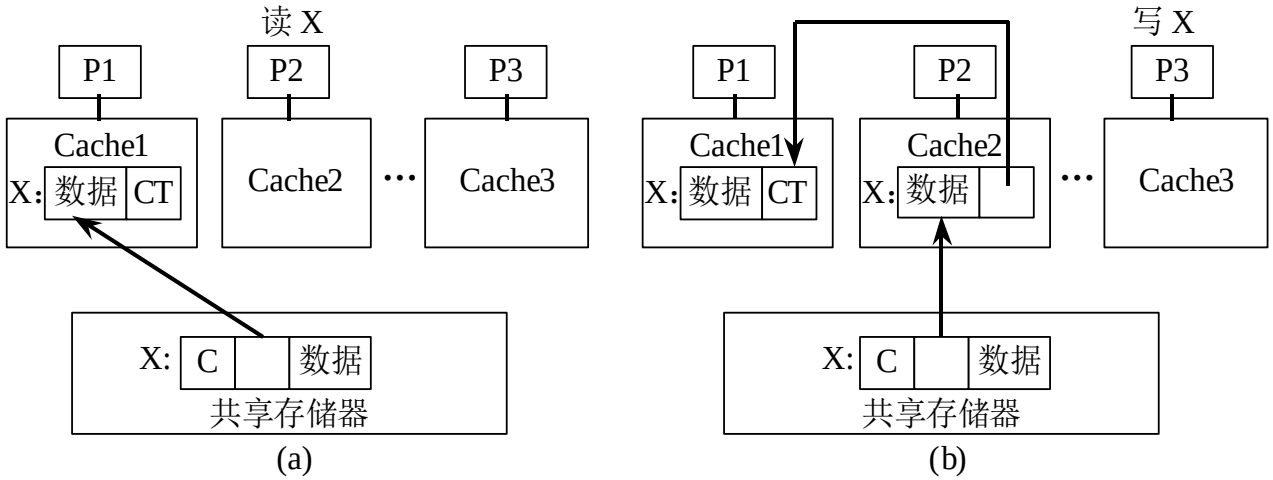


有限目录的驱逐

8.2.3 基于目录的协议

3. 链式目录

- 通过维护一个目录指针链来跟踪共享数据拷贝。
- 当P1读X时，存储器送X到Cache1,同时写Cache1的一个链结束指针CT，在存储器中也保存一个指向Cache1的指针。
- 当P2读X时，存储器送X给Cache2，同时给Cache2一个指向Cache1的指针,存储器则保存一个指向Cache2的指针。
- 当某一处理机需要写X时,它必须沿整个目录链发送一个数据无效信息。在收到所有处理机的回答信号之后，存储器才给该处理机写允许权。



链 式 目 录

8.2.3 基于目录的协议

3. 链式目录

- 当Cache中的数据块需要替换时，要把该Cache从目录链中删除。有如下解决办法：

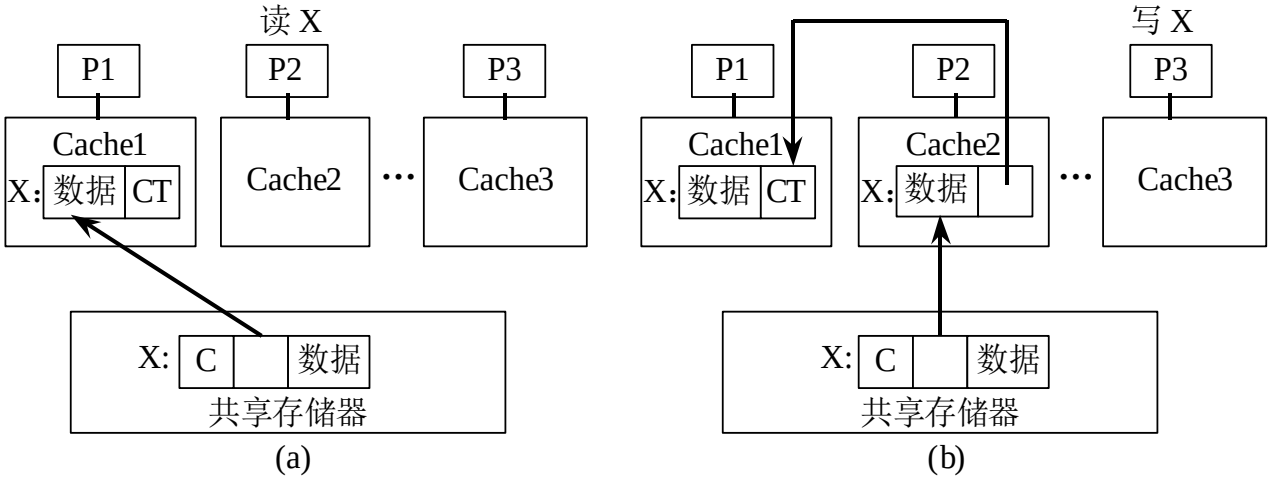
  1） 把$Cache_{i+1}$的指针指向$Cache_{i-1}$。在$Cache_i$中存放新数据块。

  2） 使$Cache_i$及在链中位于其后的所有Cache中的单元X无效。

  3） 使用双向链。在替换时不再需要遍历整个链。但指针增加了一倍，一致性协议也更加复杂。

8.2.3 基于目录的协议

3. 链式目录



链 式 目 录

- 优点：不限制共享数据块的拷贝数目，又保持了可扩展性。指针的长度以处理机数目的对数关系增长，Cache的每个数据块的指针数目与处理机数目无关。
- 缺点：链式目录的复杂程度超过了前两种目录。

8.3 多处理机的并行性和性能

8.3.1 并行算法

- **按运算基本对象：**
  - 数值型：基于代数运算（如矩阵运算、多项式求值、线性方程组求解等）
  - 非数值型：基于关系运算（如选择、排序、查找、字符处理等）
- 按并行进程间的操作顺序不同
- 按计算任务的大小

8.3 多处理机的并行性和性能

8.3.1 并行算法

- 按运算基本对象：
- **按并行进程间的操作顺序不同**
  - 同步型：并行的各进程间由于相关，必须顺次等待
  - 异步型：各进程间执行时相互独立，不会因相关而等待，只是根据执行情况决定中止或者继续
  - 独立型：并行的各进程间完全独立，进程间不需要相互通信
- 按计算任务的大小

8.3 多处理机的并行性和性能

8.3.1 并行算法

- 按运算基本对象：
- 按并行进程间的操作顺序不同
- **按计算任务的大小**
  - 细粒度：向量或循环级的并行
  - 中粒度：较大的循环级并行，确保这种并行的好处可以补偿并行带来的额外开销
  - 粗粒度：子任务间的并行

8.3 多处理机的并行性和性能

8.3.1 并行算法

➢将大的程序分解成可由足够的并行处理的过程（进程、任务、程序段）

- 每个过程被看成是一个**结点**，将过程之间的关联关系用结点组成的**树**来描述。
- 程序内各过程之间的关系就可被当成是一种算术表达式中各项之间的运算，表达式中的每一项都可看成是一个程序段的运行结果
- 研究程序段之间的并行问题就可设想成是**对算术表达式如何并行运算的问题**。

8.3 多处理机的并行性和性能

8.3.1 并行算法

➢将大的程序分解成可由足够的并行处理的过程（进程、任务、程序段）

- 每个过程被看成是一个**结点**，将过程之间的关联关系用结点组成的**树**来描述。

➢性能效率的表示

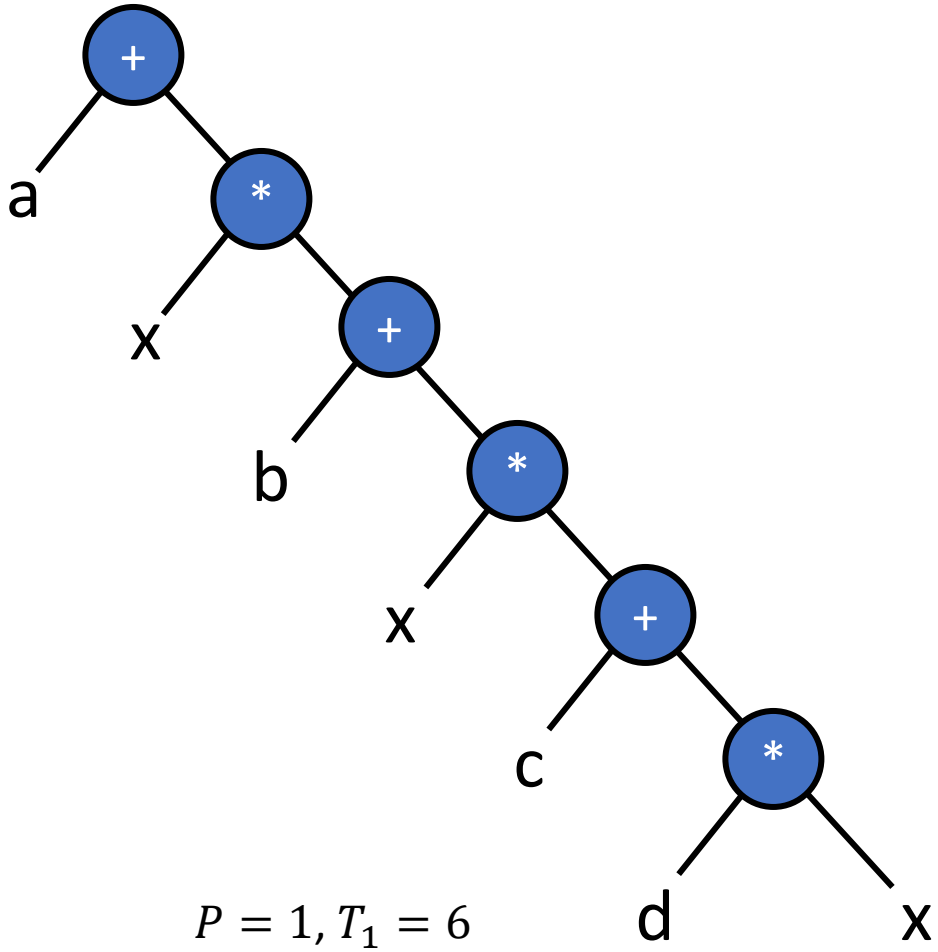
- P：可并行处理的处理机数量
- T_P：表示P台处理机运算的级数（即树高）
- S_P：加速比，即单处理机顺序运算的级数T_1与P台处理机并行运算的级数T_P之比
- E_P：P台处理机的设备利用率，$E_P = S_P/P$

8.3 多处理机的并行性和性能

8.3.1 并行算法

- E1=a+bx+cx^2+dx^3
- 利用Horner法：

  E1=a+x(b+x(c+x(d)))
- 需3个乘加循环，6级运算
- 适合于单处理机
- 用树形流程图

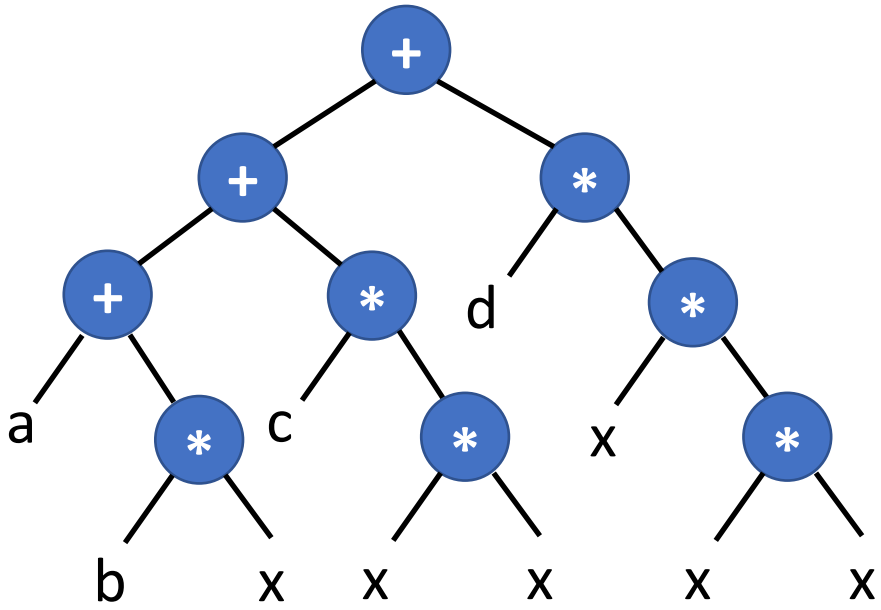


$P = 1, T_1 = 6$

8.3 多处理机的并行性和性能

8.3.1 并行算法

- $E1=a+bx+cx^2+dx^3$
- 用3台处理机，需4级运算

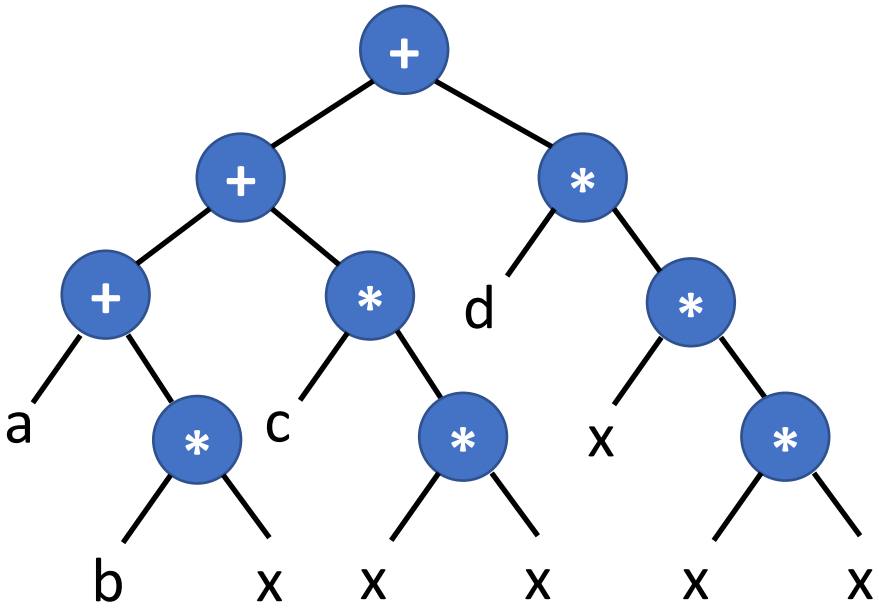


$$P=3, T_P=4, S_P=\frac{6}{4}=\frac{3}{2}, E_P=\frac{1}{2}$$

8.3 多处理机的并行性和性能

8.3.1 并行算法

- 降低树高，增加并行性
- 用交换律、结合律、分配律来变换
- 先利用交换律把相同的运算集中起来，再用结合律把参加运算的操作数配对，尽可能并行运算，最后再把子树结合起来

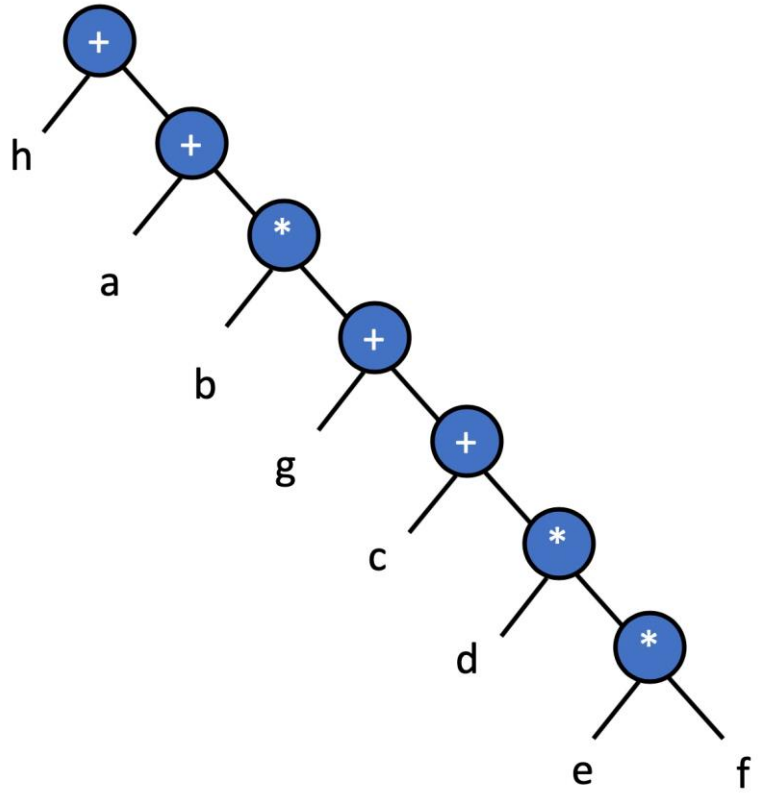


$$P = 3, T_P = 4, S_P = \frac{6}{4} = \frac{3}{2}, E_P = \frac{1}{2}$$

8.3 多处理机的并行性和性能

8.3.1 并行算法

- 计算$E_2 = a + b(c + edf + g) + h$

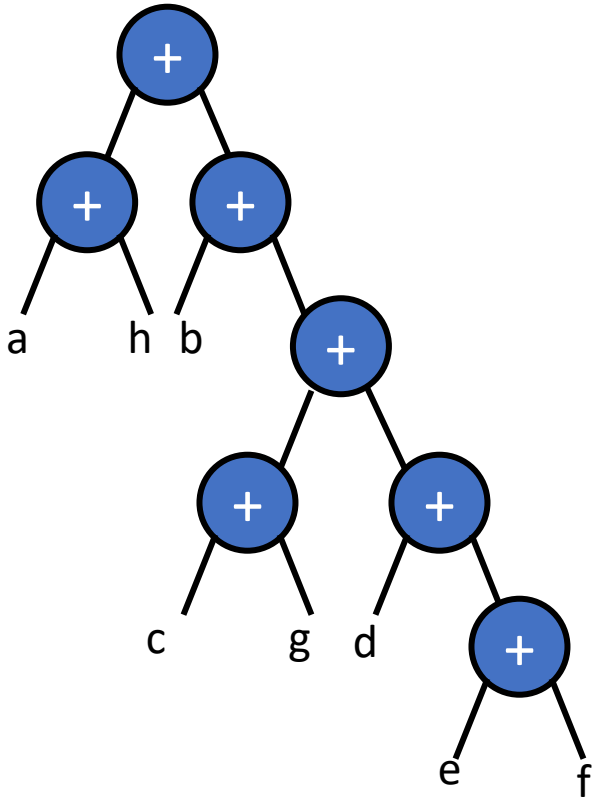


8.3 多处理机的并行性和性能

8.3.1 并行算法

- 计算$E_2 = a + b(c + edf + g) + h$
- 利用交换律和结合律

$$E_2 = (a + h) + b\big((c + g) + def\big)$$



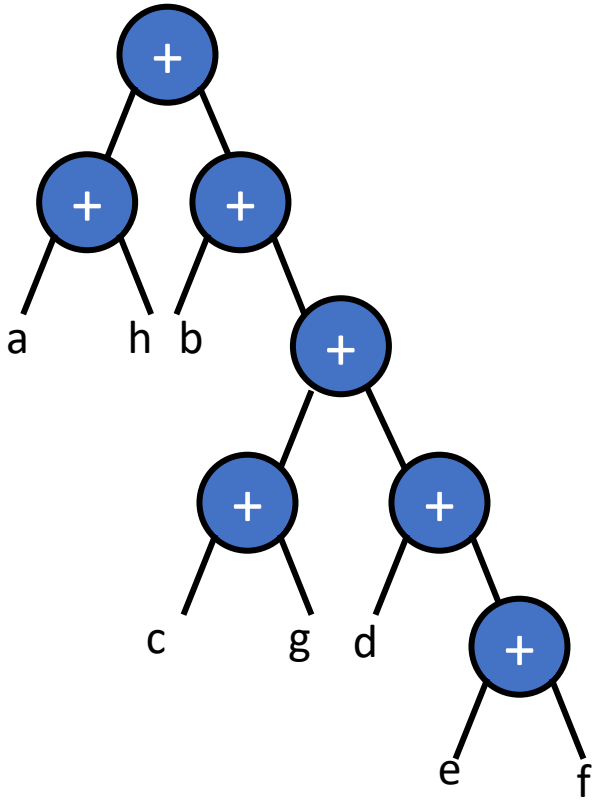
$$P = 2, T_P = 5, S_P = 1.4, E_P = 0.7$$

8.3 多处理机的并行性和性能

8.3.1 并行算法

- 计算$E_2 = a + b(c + edf + g) + h$
- 利用交换律和结合律

$$E_2 = (a + h) + b\big((c + g) + def\big)$$



$$P = 2, T_P = 5, S_P = 1.4, E_P = 0.7$$

8.3 多处理机的并行性和性能

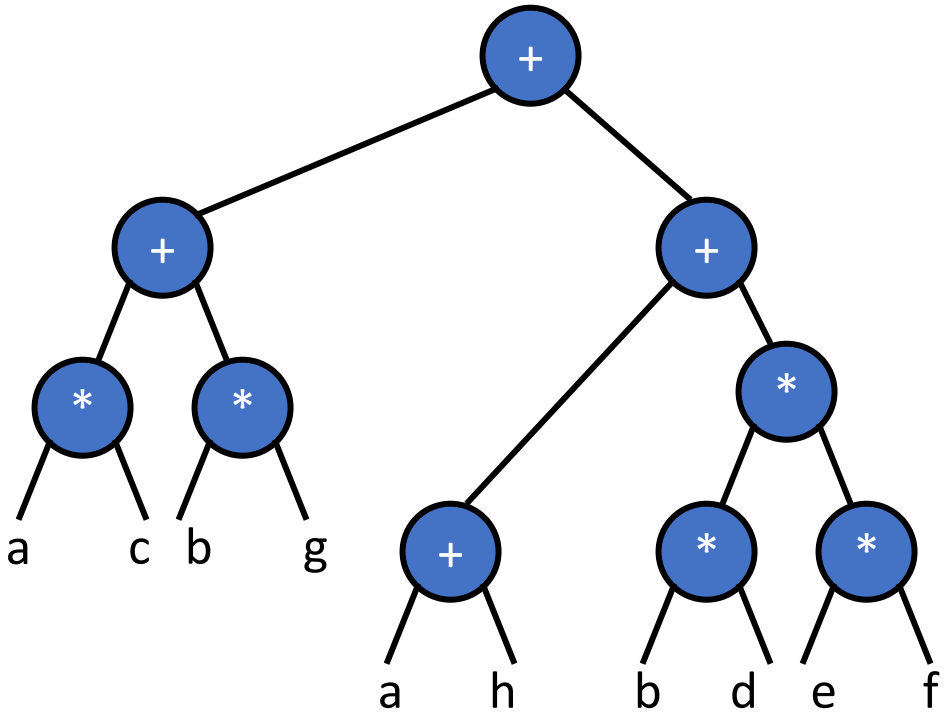
8.3.1 并行算法

- 计算$E_2 = a + b(c + edf + g) + h$
- 利用交换律和结合律

$$E_2 = (a + h) + b\big((c + g) + def\big)$$

- 利用分配律，进一步降低树高

$$E_2 = (a + h) + bc + bg + bdef$$



$$P = 3, T_P = 4, S_P = 7/4, E_P = 7/12$$

8.3 多处理机的并行性和性能

8.3.2 程序并行性分析

- 假定一个程序包含$P_1, P_2, \cdots, P_i, \cdots, P_j, \cdots, P_n$等$n$个程序段，设$P_i$和$P_j$程序段都是一条语句，$P_i$在$P_j$之前执行。
  - 数据相关
  - 数据反相关
  - 数据输出相关

8.3 多处理机的并行性和性能

8.3.2 程序并行性分析

➢ 数据相关

- 如果P_i的左部变量在P_j的右部变量集内，且P_j必须取出P_i运算的结果来作为操作数，称P_j数据相关于P_i，例：

P_i： A=B+D

P_j： C=A*E

- 相当于流水中的“先写后读”相关。顺序执行的正确结果是：

P_i $A_{新}=B_{原}+D_{原}$

P_j $C_{新}=A_{新}*E_{原}=(B_{原}+D_{原})*\ E_{原}$

8.3 多处理机的并行性和性能

8.3.2 程序并行性分析

➢ 数据相关

- P_i，P_j不能并行。
- 特殊，当P_i和P_j服从交换律时

P_i　　A=2*A

P_j　　A=3*A

- 虽不能并行，但能交换串行，得到6*A

8.3 多处理机的并行性和性能

8.3.2 程序并行性分析

➢ **数据反相关**

- 如果P_j的左部变量在P_i的右部变量集内，且当P_i未取用其变量的值之前，是不允许被P_j所改变，称P_i**数据反相**关于P_j，例：

  P_i：　C=A+E

  P_j：　A=B+D

- 相当于流水中的“先读后写”相关。顺序执行的正确结果是：

  P_i　　$C_{新}=A_{原}+E_{原}$

  P_j　　$A_{新}=B_{原}+D_{原}$

  不能交换串行。

8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

- 是在普通顺序型程序设计语言基础上加以扩充，增加能明确表示并行进程的成分。
- 并行程序设计语言要求能使程序员灵活方便地在其程序中表示出各类并行性，同时应有高的效率，能在各种并行/向量计算机系统中有效地实现。
- 并行进程的特点：进程在时间上重叠执行。
- 派生：FORK;　汇合：JOIN
- 在不同的机器上有不同的表现形式。

8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

- M.E.Conway(康佳)形式：
  - FORK m：派生出标号为m开始的新进程。
    - 如果是共享内存，产生存储器指针、映像函数和访问权数据
    - 将空闲的处理机分配给被FORK语句派生的新进程
    - 如果没有可用的空闲处理机，排队等待。

8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

- M.E.Conway(康佳)形式：
  - JOIN m ：
    - 附有计数器，初值为0
    - 执行语句时，计数器加1，与n比较
    - 如相等，表明执行的第n格并发进程经过JOIN语句，允许通过语句，计数器清0，继续执行后续语句。
    - 如小于n，则进程不是最后一个，先让进程结束，则把它占用的处理机释放出来，分配给排队的其他任务，如无任务，则空闲

8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

- 计算 Z=E+A*B*C/D+F
- 并行编译：

S1　　G=A*B

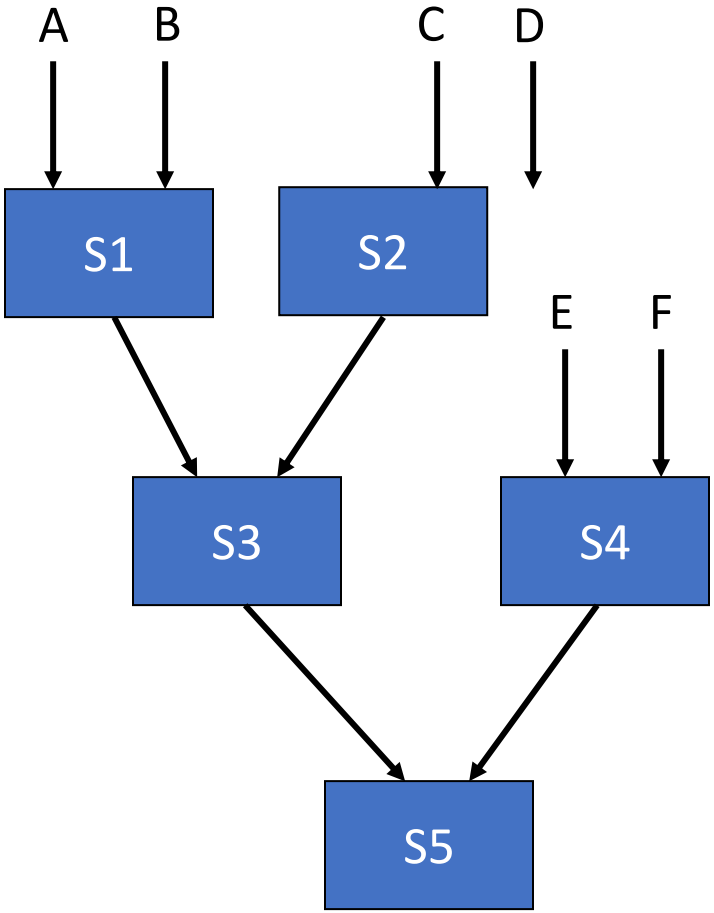
S2　　H=C/D

S3　　I=G*H

S4　　J=E+F

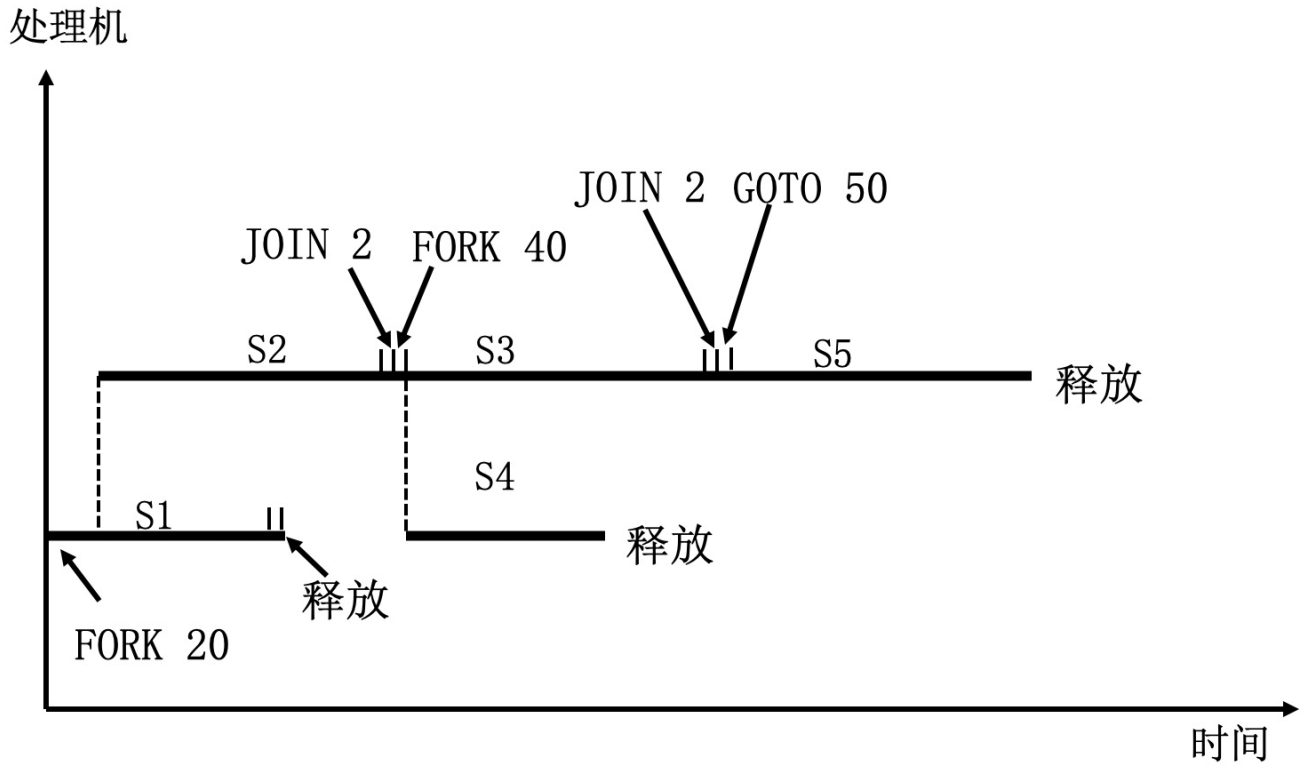
S5　　Z=I+J

```
      FORK  20
10  G=A*B  (进程S1)
      JOIN 2
      GOTO 30
20  H=C/D  (进程S2)
      JOIN 2
30  FORK  40
      I+=G*H (进程S3)
      JOIN  2
      GOTO 50
40  J=E+F   (进程S4)
     JOIN 2
50  Z=I+J   (进程S5)
```



8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

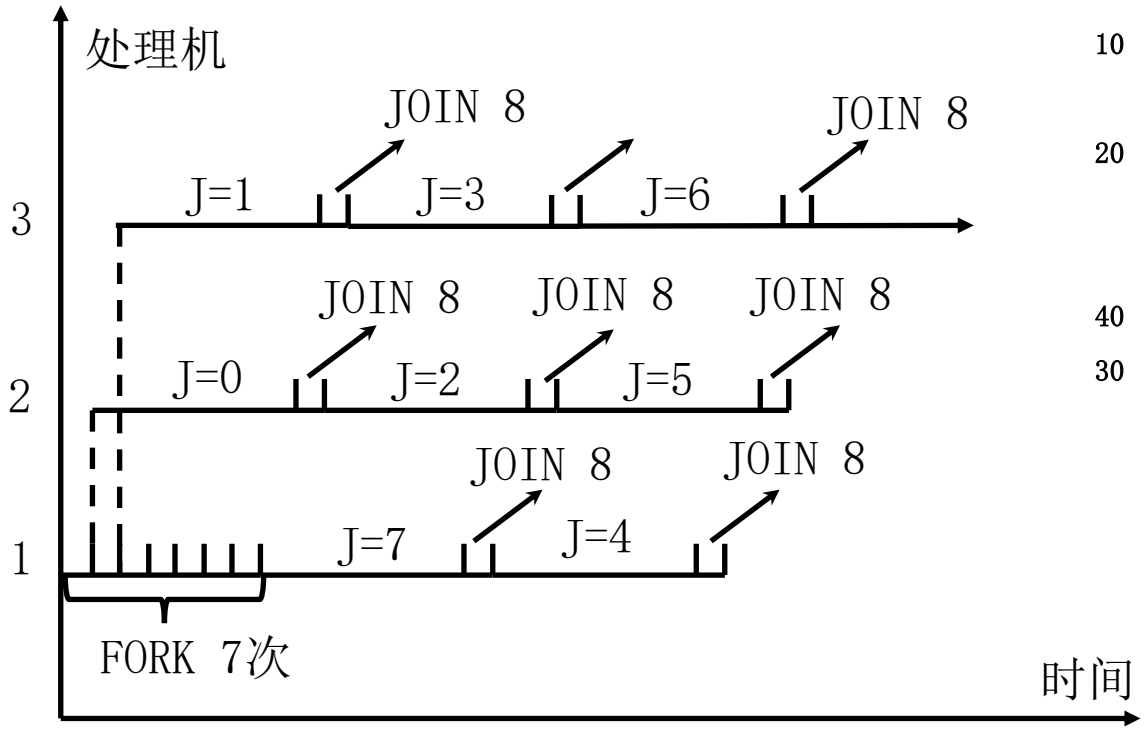



```
      FORK 20
10  G=A*B (进程S1)
      JOIN 2
      GOTO 30
20  H=C/D (进程S2)
      JOIN 2
30  FORK 40
      I+=G*H (进程S3)
      JOIN 2
      GOTO 50
40  J=E+F   (进程S4)
      JOIN 2
50  Z=I+J   (进程S5)
```

8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

- 实现两个8*8矩阵相乘




```
      DO 10 J=0, 6
10    FORK 20
      J=7
20    DO 30 I=0, 7
      C(I, J)=0
      DO 40 K=0, 7
40    C(I, J) = C(I, J)+A(I, K)*B(K, J)
30    CONUNE
      JOIN 8
```

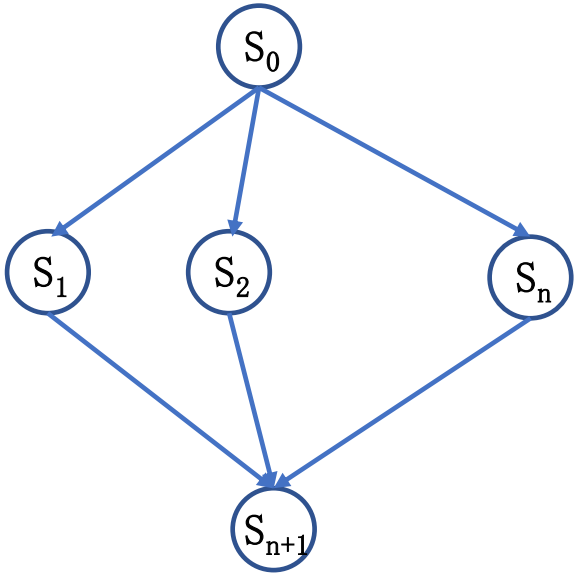
8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

➢ E.W. Dijkstra将FORK-JOIN概念进一步发展，提出了一种等价的块结构式语言

```
begin
S0;
Cobegin S1; S2; … ; Sn; Coend
Sn+1;
end
```

```
begin
S0;
parbegin S1; S2; … ; Sn; parend
Sn+1;
end
```



- 语句S_i进一步发展改变的变量只属于该进程专用，而不能被其他并行语句引用；它们可以使用但不能修改共享变量，能修改只属于本进程的局部变量

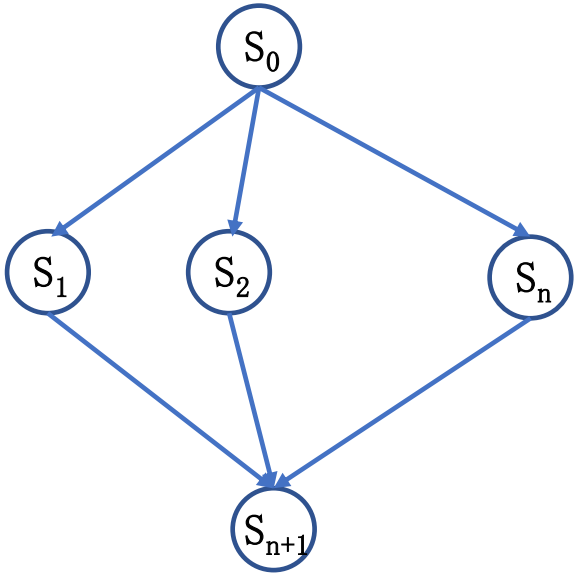
8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

➢ E.W. Dijkstra将FORK-JOIN概念进一步发展，提出了一种等价的块结构式语言

```
begin
S0;
Cobegin S1; S2; … ; Sn; Coend
Sn+1;
end
```

```
begin
S0;
parbegin S1; S2; … ; Sn; parend
Sn+1;
end
```



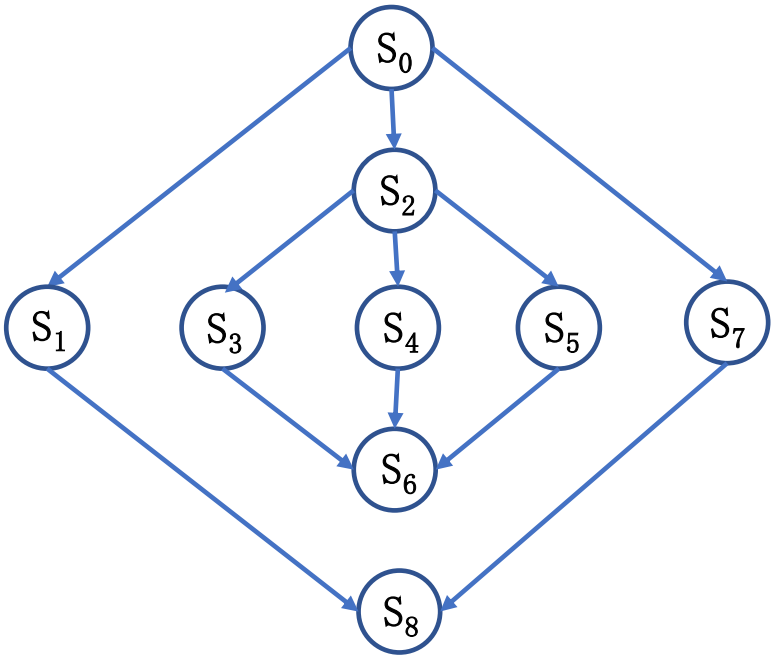
- 语句S_i进一步发展改变的变量只属于该进程专用，而不能被其他并行语句引用；它们可以使用但不能修改共享变量，能修改只属于本进程的局部变量

8.3 多处理机的并行性和性能

8.3.3 并行语言与并行编译

- E.W. Dijkstra将FORK-JOIN概念进一步发展，提出了一种等价的块结构式语言

begin
S_0;
Cobegin
S_1;
begin
S_2
Cobegin S_3; S_4; S_5; Coend
S_6;
end
S_7;
Coend
S_8;
end



嵌套并行进程的优先执行过程