

# 第七章 多处理器

Dr. Feng Li

[fli@sdu.edu.cn](mailto:fli@sdu.edu.cn)

<https://funglee.github.io>

# 多处理器的概念

- 多处理器是指两台以上的处理机，共享I/O子系统，机间经共享主存或高速通信网络通信，在统一操作系统的控制下，协同求解大而复杂问题的计算机系统
- 主要目的
  - 相比阵列处理机主要实现向量指令操作级的并行，利用开发并行性中的同时性，多处理器系统通过多台处理机对多个作业、任务进行并行执行来提高解题速度，从而提高系统的整体性能，利用的是并行性中的并发性
  - 使用冗余的多个处理机通过重新组织来提高系统的可靠性和适应性

- 多处理机是一种多指令多数据流系统
  - 在硬件结构上，多个处理机要用多个指令部件分别控制，通过共享主存或机间互联网络实现异步通信
  - 在算法上，不限于向量和数组的处理，还要挖掘和实现更多通用算法中隐含的并行性
  - 在系统管理上，更多依靠操作系统等软件手段，有效解决资源分析和管理

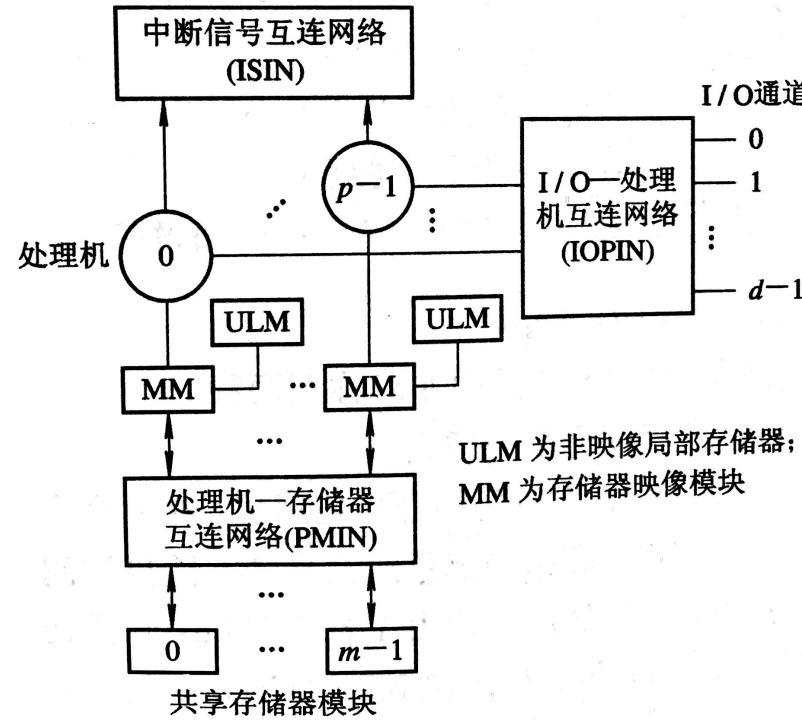
# 多处理机要解决的问题

- 在硬件结构上要解决好处理机、存储器模块和I/O子系统之间的互连关系，提高带宽、降低成本、实现机间通信的多样性、灵活性和不规则性，又要避免资源争用，实现无冲突连接
- 最大限度地通过多种渠道开发系统的并行性，提高系统性能
- 如何使用专门的指令和语句来控制并行任务的派生
- 对一个大的作业和任务采用合适的粒度，进行分割和聚合，提高并行度，降低开销
- 动态的资源分配和任务调度，实现处理机负载均衡，防止死锁
- 系统的容错性和可恢复性
- 为用户提供良好的编程环境

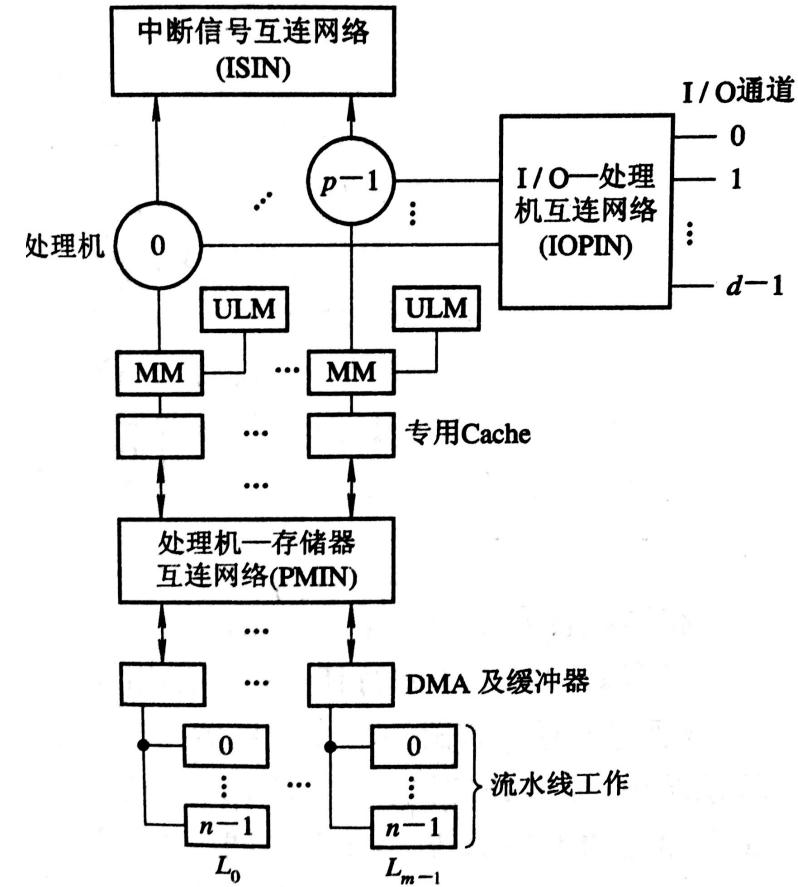
# 多处理机硬件结构：紧耦合多处理机

- 通过共享主存实现处理机之间的通信，因此通信速率受限于主存频宽
- 各个处理机通过互连网络与主存相连
- 处理机数量受限于互连网络带宽以及各处理机访主存冲突的概率
- 系统组成
  - $p$ 台处理机
  - $m$ 个存储器模块（模 $m$ 多体交叉存取）
  - $d$ 个I/O通道
  - “处理机—存储器”互连网络（PMIN）
  - “I/O—处理机”互连网络（IOPIN）
  - 中断信号互连网络（ISIN）

# 紧耦合多处理机的两种构形



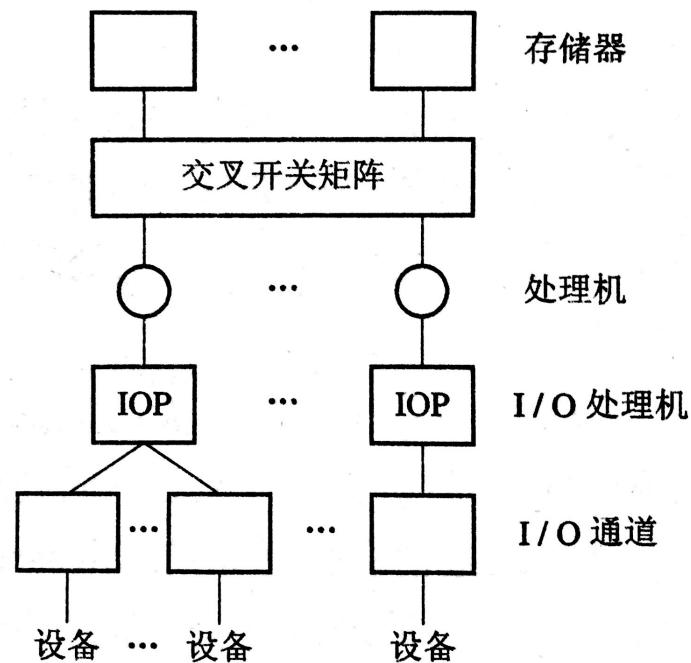
处理机不带专用Cache



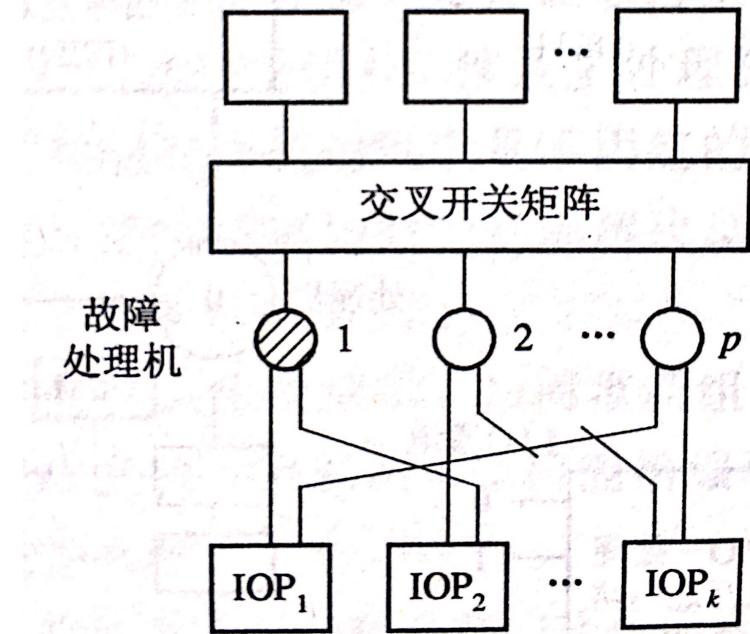
处理机自带专用Cache

- “处理机—存储器”互连网络实现处理机和存储器之间的连接
  - 通过仲裁机制，每个存储器模块在一个存储周期只响应其中一台处理机的访存请求
  - 为减少多个处理机同时访问同一存储器模块的冲突， $m \geq p$
  - 为减少访主存信息量和“处理机—存储器”互连网络使用冲突，降低访存冲突，可以设置局部存储器
  - 处理机可自带Cache，以减少访主存次数，进一步降低访存冲突
- 通过中断信号互连网络，一台处理机可以向另一台处理机发送中断信号，实现处理机间的进程同步
- 处理机和连接外设的I/O可以通过“I/O—处理机”互连网络进行完全连接的对称的通信

- 对称的“I/O—处理机”互连网络虽然连接灵活，但是成本较高，因此多数多处理机采用非对称互连
- 在非对称的I/O子系统中，为防止某台处理机失效时所连接的外设无法连接到其他处理机，应采取适当的冗余



带非对称I/O子系统的多处理机结构



采用冗余连接的非对称I/O子系统

# 多处理机硬件结构：松耦合多处理机

- 每台处理机都有一个容量较大的局部存储器，用于存储经常用的指令和数据
- 不同处理机间通过通道互连或者消息传送系统（Message Transfer System, MTS）实现通信，以共享外部设备和交换信息
- 松耦合多处理机适合做粗粒度的并行计算
- 任务之间相对独立，其信息流量较少

# 机间互连形式

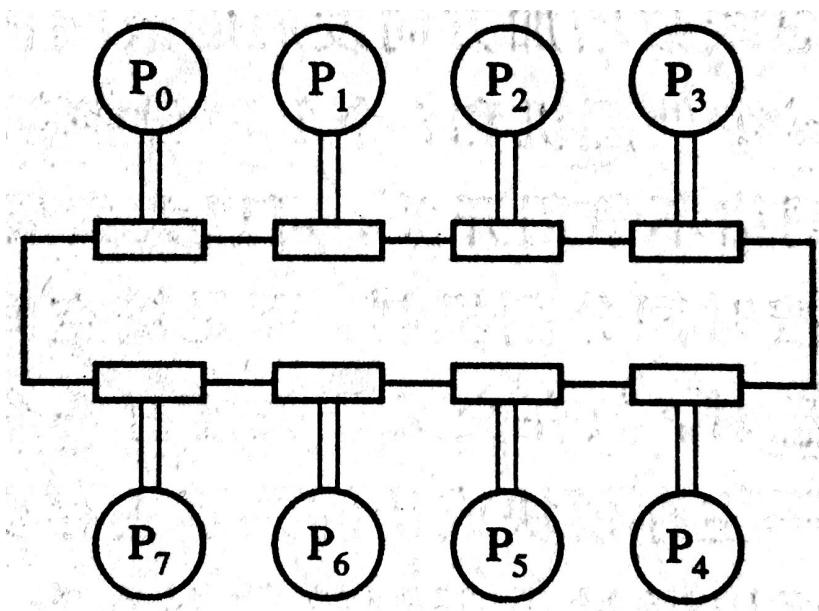
- 多个处理机、存储器模块和外围设备通过接口与公用总线相连，采用分时或多路转接技术传送
- 单总线结构简单，成本低，扩展方便，但是对总线的失效敏感，处理机数量的增加会增大总线冲突概率，使系统效率急剧下降
- 可采用多总线机制减少总线访问冲突的概率

# 解决多处理机同时访问公用总线冲突

- 静态优先级算法：为每个连到总线的部件分配一个固定的优先级，当多个部件同时请求时，优先级高的部件先使用总线
- 固定时间片算法：把总线按固定大小的时间片轮流提供给部件使用
- 动态优先级算法：总线上各个部件的优先级根据情况按一定规则动态改变
- 先来先服务算法：按接收到访问总线请求的先后顺序来响应

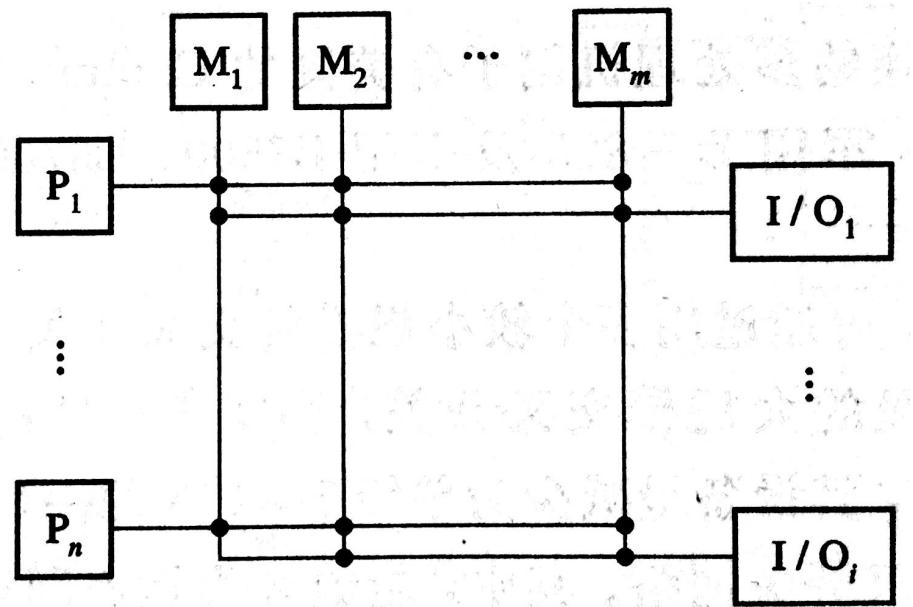
# 环形互连形式

- 发送进程将信息送到环上，经环形网络不断向下一台处理机传递，直到此信息又回到发送者处为止
- 在环上循环播送令牌（Token），持有令牌的处理机可以发送信息



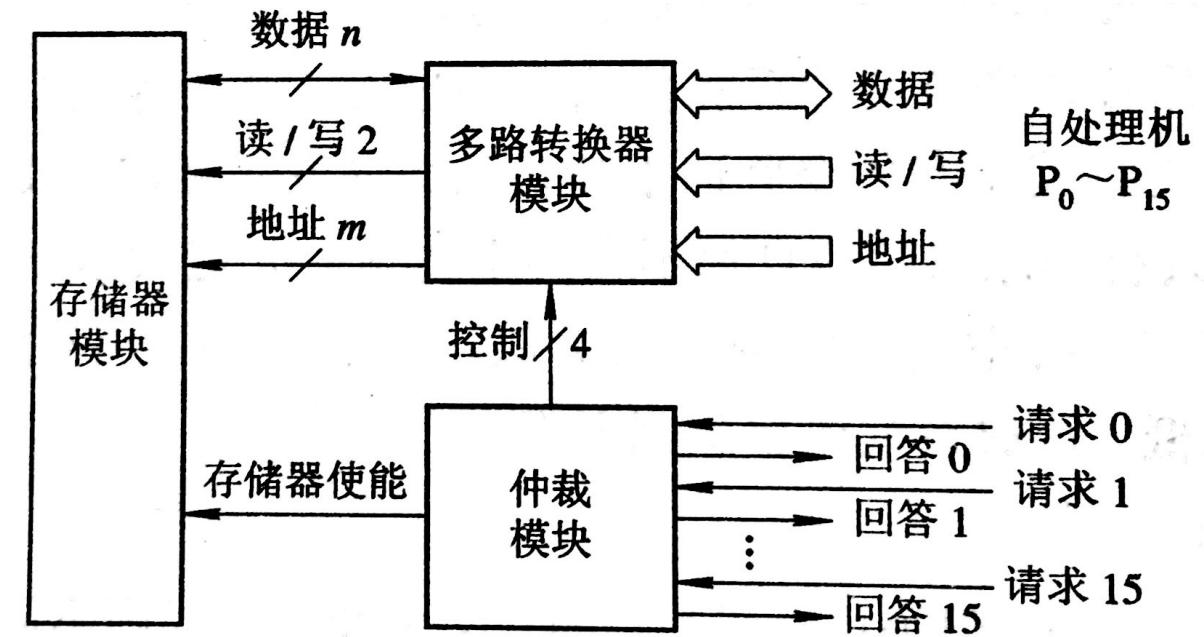
# 交叉开关形式

- 纵横开关阵列将横向的处理机及I/O通道与纵向的存储器模块连接起来
- 总线数= $n+i+m$
- 低延迟、高带宽
- 多个处理机或I/O通道访问同一主存模块或访问共享主存变量仍然会产生访存冲突



# 举例：节点开关的结构 (C. mmp)

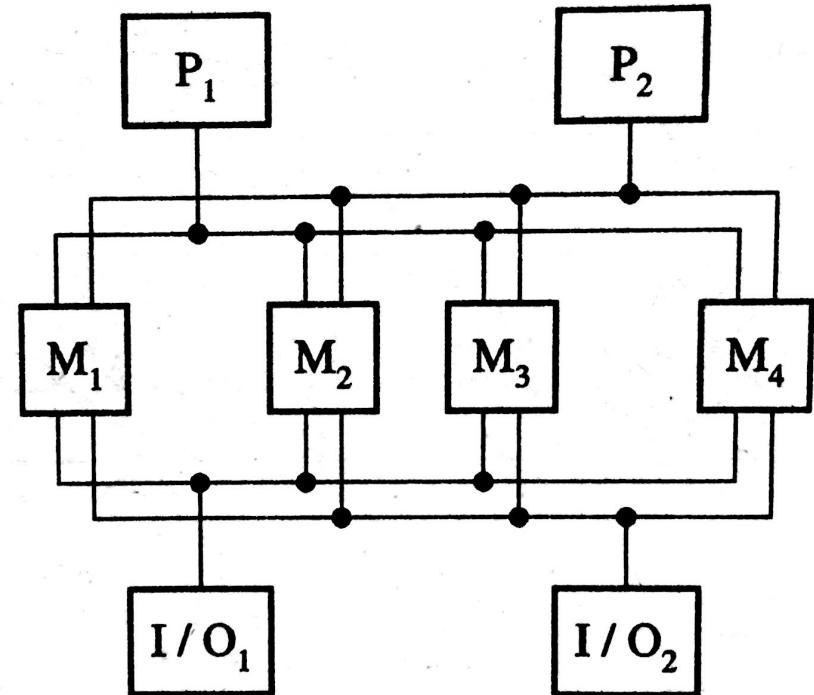
- 由仲裁模块和多路转换器模块组成
- 16个处理机都可以给仲裁模块发送访存请求
- 仲裁模块根据一定的算法，响应有最高优先级的处理机请求，并返回该处理机一个回答
- 处理机接到回答，通过多路转换器模块进行访存



交叉开关中结点开关的结构

# 多端口存储器形式

- 每个存储器模块都有多个访问端口
- 将交叉开关矩阵中的控制、转移和优先级仲裁等功能转移到相应的存储器模块的接口中
- 端口数目不宜太多，一经确定就不易改变

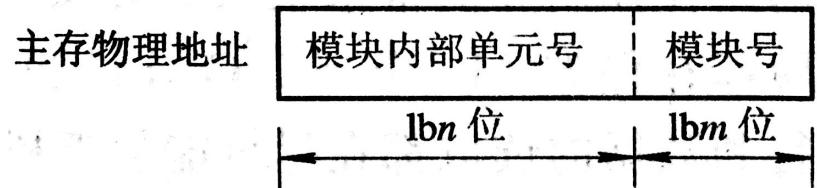
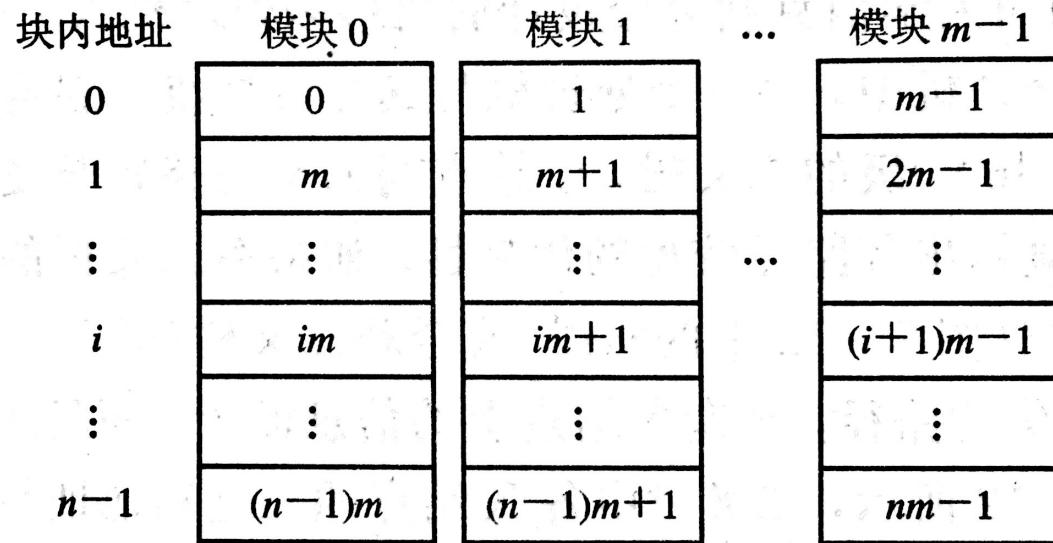


四端口存储器形式的结构

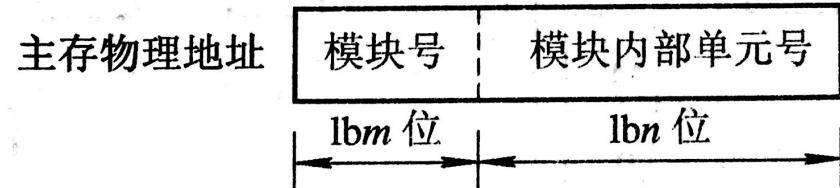
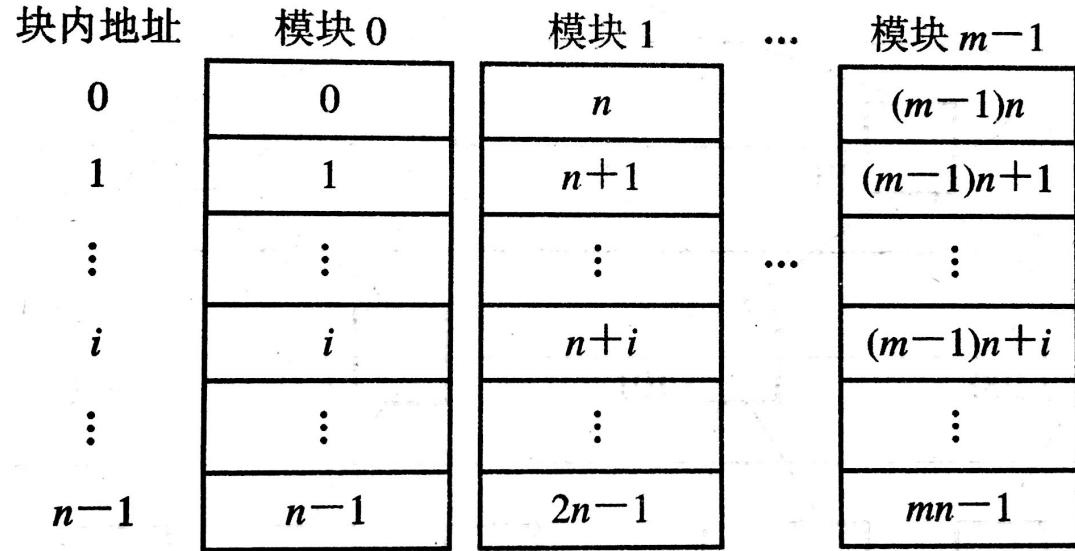
# 其它形式

- 蠕虫穿洞寻径网络：处理机间设置小容量缓冲存储器，以实现消息分组的寻径存储转发
- 开关枢纽结构形式：把互连结构的开关设置在各处理机或接口内部，组成分布式结构

# 存储器组织

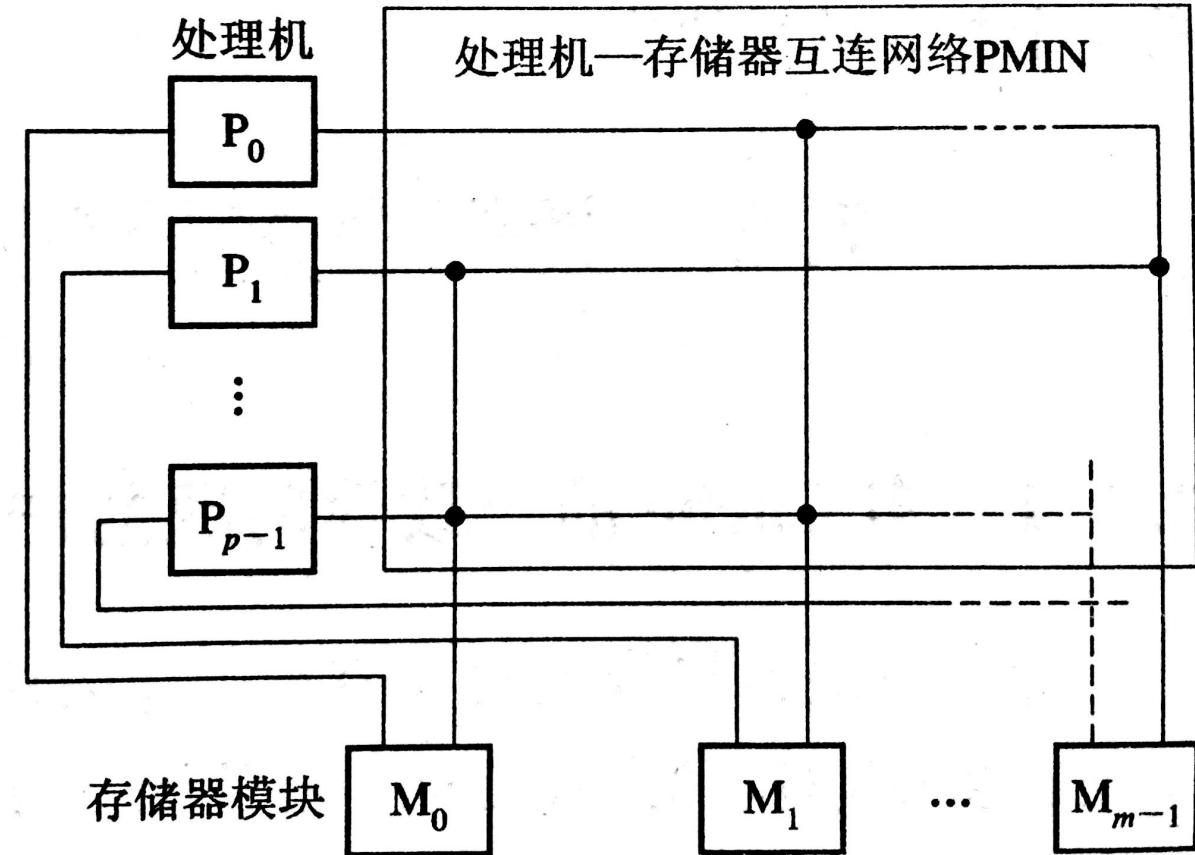


m个模块的低位交叉编址



m个模块的高位交叉编址

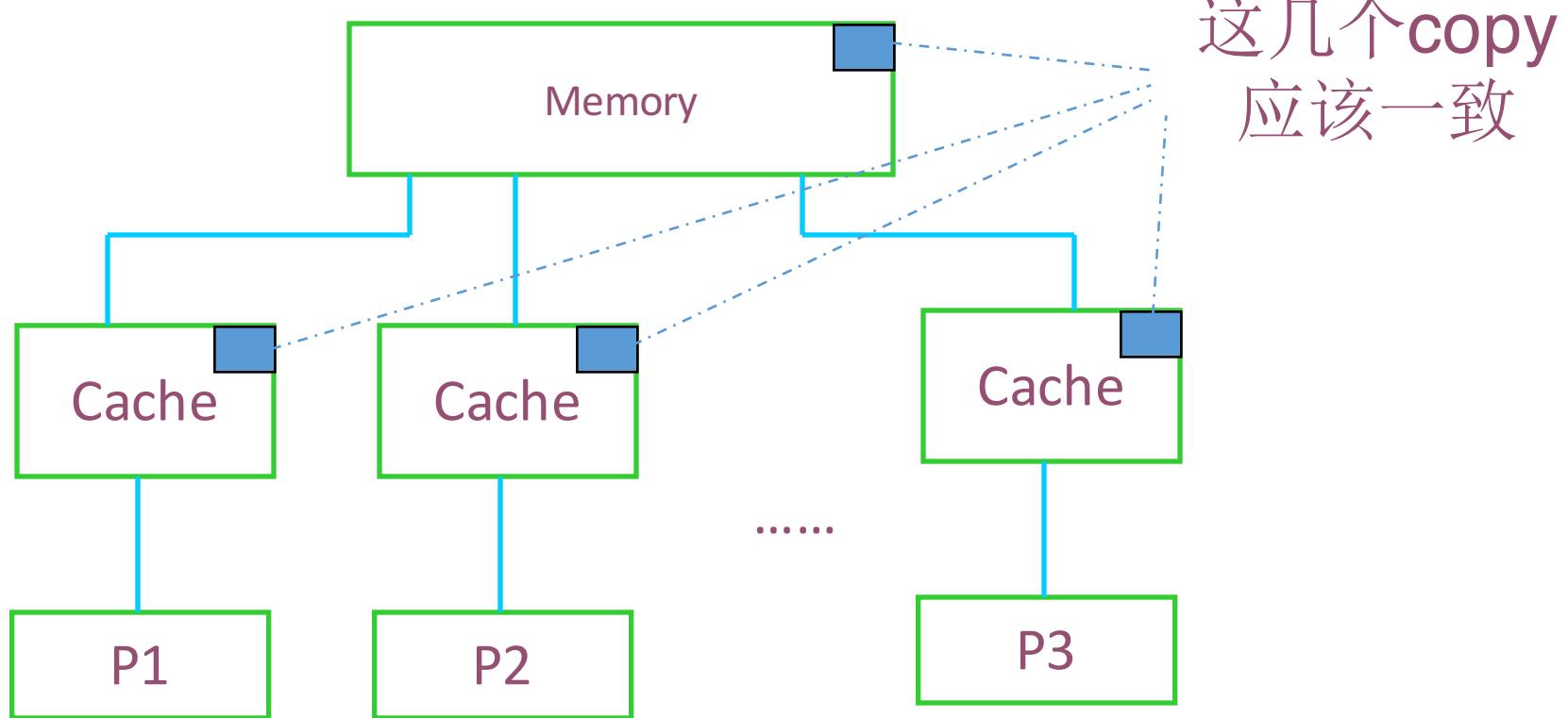
- 当各个处理机上活跃的进程是共享同一集中连续物理地址空间中的数据时，主存采用低位交叉编址
- 当处理机较少或基本不共享集中的数据时，低位交叉编址易引起访存冲突，宜采用高位交叉编址
- 高位交叉编址中，可以在给定存储器模块中为处理机*i*进程集中一定数量的页面，以减少访存冲突，该存储器模块称为处理机*i*的本地存储器（Home Memory）
- 延伸：当系统存储器模块多于处理机数量时，处理机*i*的本地存储器可以动态地由这些存储模块中的某几个集合组成；而且，任何时候，每个存储器模块只被一个处理机所访问



本地存储器概念

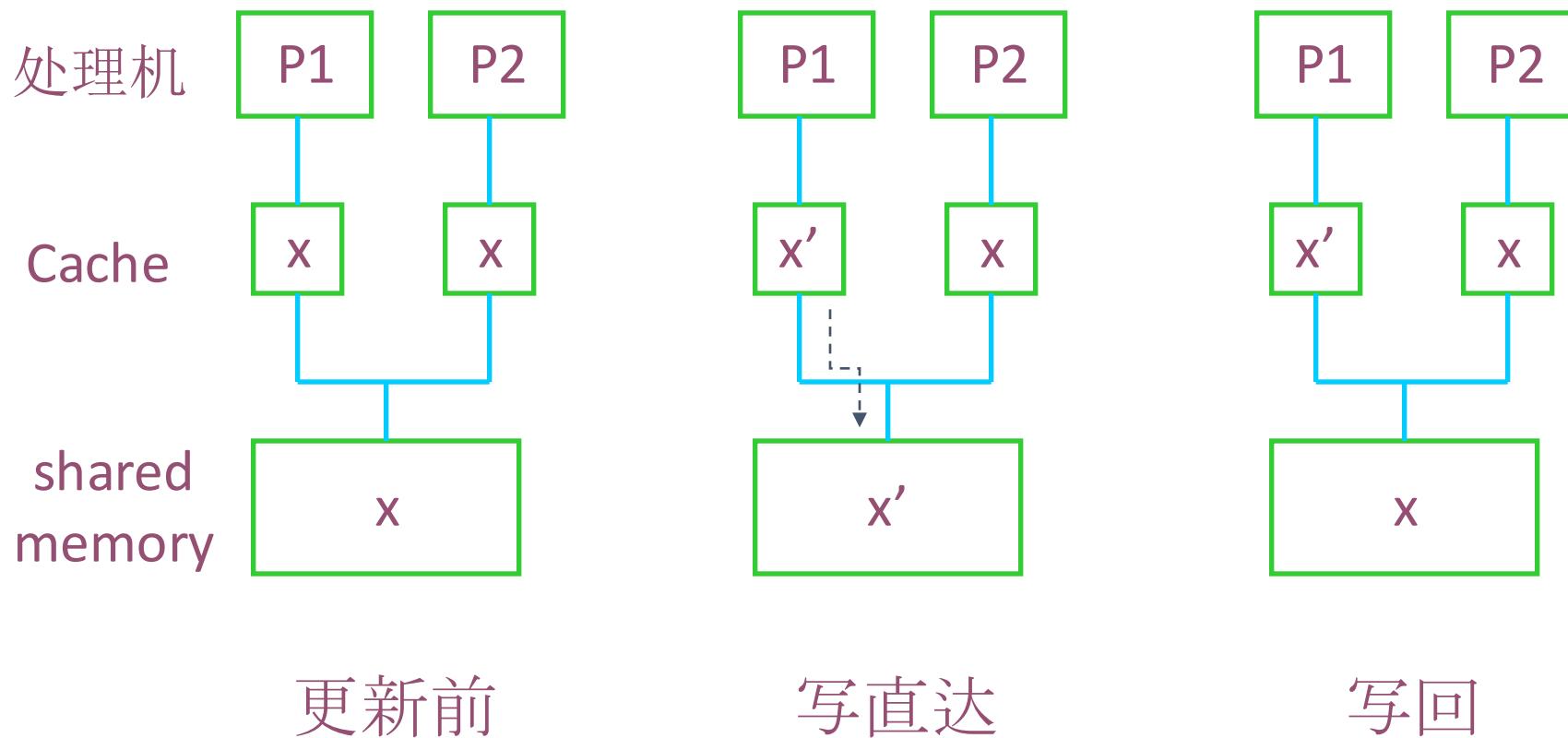
# 多Cache的一致性问题

- 每个处理机都有自己专用的Cache，当主存中同一个信息块在多个Cache中都有时，会出现多个Cache之间的相应的信息块内容不一致的问题



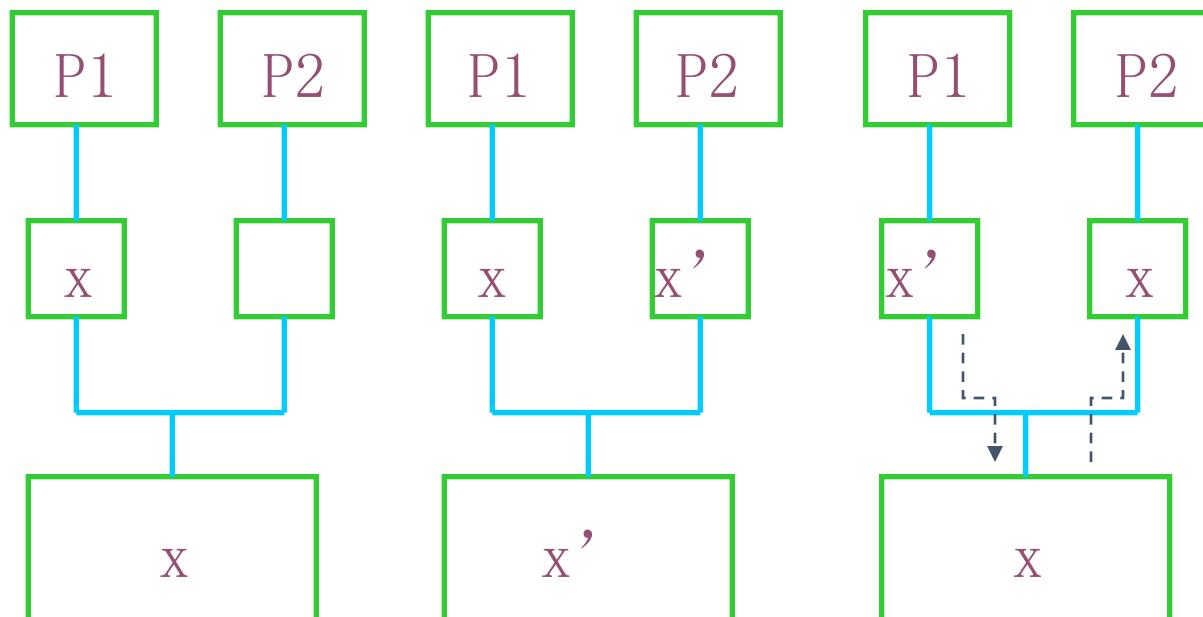
# 多Cache不一致的原因

- 共享可写数据的不一致性



# 多Cache不一致的原因

- 进程迁移的不一致性



迁移前

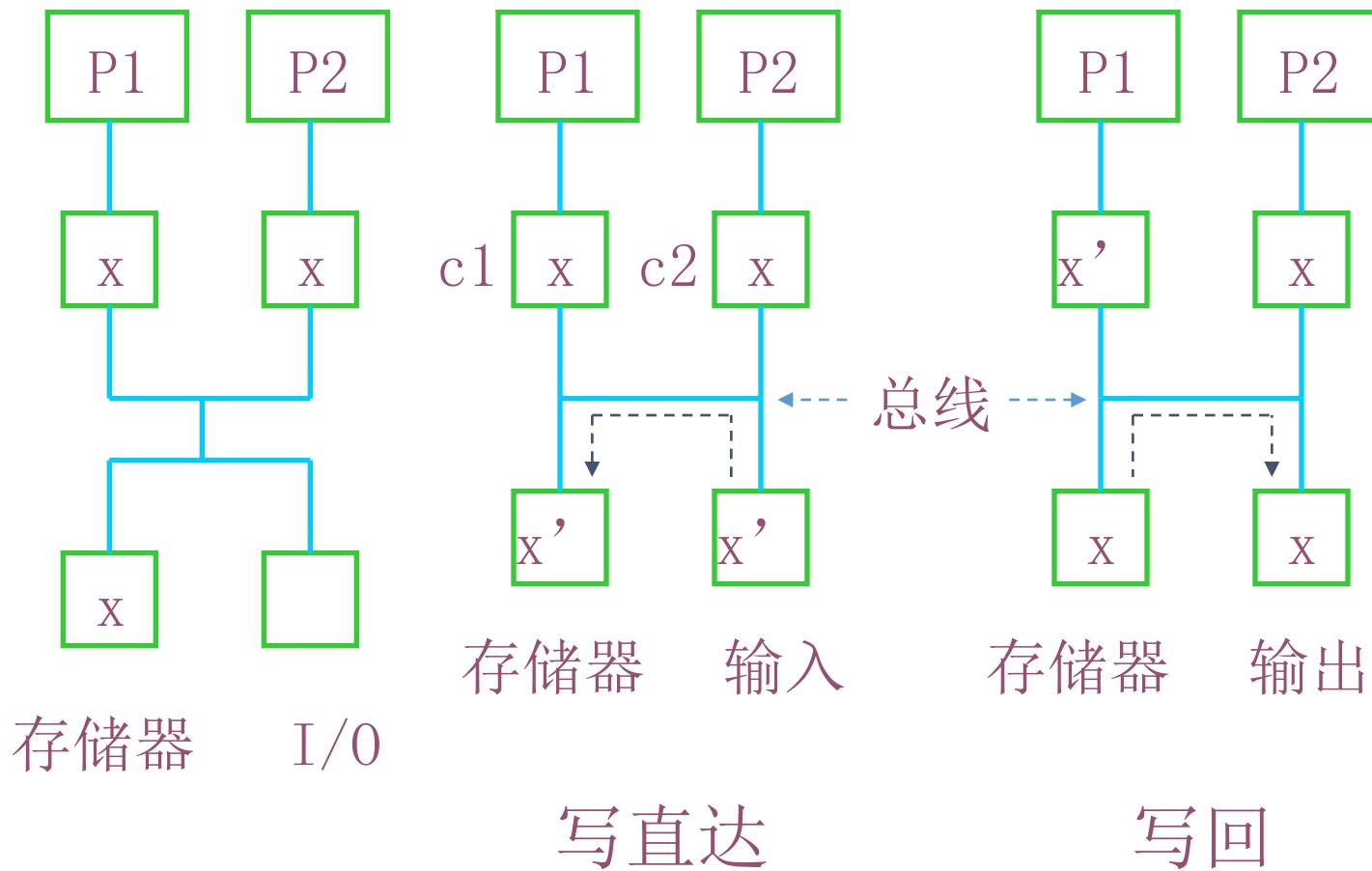
写直达

写回

- 右图为：包含共享变量x的进程原来在P1上运行，并对x进行了修改（但采取写回策略，所以暂时没有修改Memory），由于某种原因迁移到P2，修改过的x'仍在P1的Cache中，P2运行时从Memory中得到x，这个x其实是“过时”的，所以造成了不一致。
- 中间图为：P2中运行的进程对x进行了修改，采取写直达策略，所以把Memory中的x也修改为x'，由于某种原因该进程迁移到P1，但P1的Cache中仍为x，所以造成不一致。

# 多Cache不一致的原因

- 绕过Cache的I/O操作



- 中间图为：当I/O处理机将一个新的数据 $x'$ ，写入主存储器时，绕过采用写通过策略的cache，则C1和共享存储器之间产生了不一致。
- 右图为：直接从主存储器输出数据时（绕过Cache），与采用写回策略的高速缓存产生不一致性。

# 两种设计Cache一致性协议策略

- 写无效 (write invalidate)

- 任一处理机写它的私有Cache时，它都使所有其它的Cache中的副本失效
- 对Write-through，它也更新memory中的副本（最终是一个Cache中的副本和memory中的副本是有效的）。
- 对Write-back，它使memory中的副本也失效（最终只有一个Cache中的副本是有效的）

- 写更新 (write update)

- 任一处理机写它的私有Cache时，它都立即更新所有其它的Cache中的副本。
- 对Write-through，它也更新主存储器中的副本。
- 对Write-back，对存储器中副本的更新延迟到这个Cache被置换的时刻。

- 写无效的问题
  - 主要开销在两个方面：（1）作废各Cache副本的开销；（2）由作废引起缺失造成的开销，即处理机需要访问已经作废的数据时将引起Cache的缺失。
  - 如果一个处理机经常对某个块连续写，且处理机间对共享块的竞争较小，这时写无效策略维护一致性的开销是很小的。如发生严重竞争，即处理机之间对某个地址的共享数据竞争，将产生较多的作废，引起更多的作废缺失。结果是共享数据在各处理机间倒来倒去，产生颠簸现象，当缓存块比较大时，这种颠簸现象更为严重。
- 写更新的问题
  - 由于更新时，所有的副本均需要更新，开销很大。

# 监听总线协议 (Snoopy protocol)

- 通过总线监听机制实现Cache和共享存储器之间的一致性。
- 适用性分析：
  - 适用于具有广播能力的总线结构多处理机系统，允许每个处理机通过 Cache控制器监听其它处理机的存储器访问情况。
  - 只适用于小规模的多处理机系统。

# 写一次（Write-Once）协议

- 写无效监听一致性协议，将写通过和写回策略结合。
- 为了减少总线流量，高速缓存块的第一次写用写通过方法，产生一份正确的主存储器副本，并使其它的Cache中的副本无效，之后就采用写回方法更新Cache与主存储器
- 一致性协议包括：
  - Cache可能出现的状态集合
  - 共享主存的状态
  - 为维护一致性而引起的状态转换

## 每份Cache中的副本可能出现的四种状态

- **有效 (valid state)**：与主存储器副本一致的Cache副本，即该副本未经修改，所以这个Cache副本不是唯一的副本。
- **保留 (reserved state)**：这一Cache副本是第一次修改，并用写通过方法写入主存，所以这一Cache副本和主存储器副本一致。
- **重写 (dirty state)**：Cache副本不止一次被修改过，由于不再采用写通过方法，所以这个Cache副本是唯一的副本。与存储器和其它的Cache副本都不一致。主存储器中的副本也是无效的。
- **无效 (invalid state)** 与存储器或其它的Cache副本不一致，或在Cache中找不到。

- 局部命令 (Local commands)
  - P-Read: 本地处理机读自己的Cache副本。
  - P-Write: 本地处理机写自己的Cache副本。
- 一致性命令
  - Read-blk: 从另一Cache读一份有效的副本。
  - Write-inv: 在写命中时在总线上广播一个无效命令。
  - Read-inv: 在写缺失时在总线上广播一个无效命令。

# Write-Once一致性协议状态转移表

Command	Current state	Next state	Status	Action
P-Read	有效	有效	Read-hit	必是局部进行，不影响有效状态
P-Write	有效	保留	Write-hit	第一次写命中，用写通过法。同时修改本地和主存副本并广播Write-inv使所有副本失效
P-Write	保留	重写	Write-hit	第二次写命中，用写回法。但不修改主存的副本

# Write-once一致性协议状态转移表

Command	Current state	Next state	Status	Action
P-Write	重写	重写	Write-hit	第二次以后更多的写命中，用写回法
P-Write	无效	保留	Write-miss	写缺失时，则从主存或远程Cache送来副本，广播Read-inv使所有其它副本无效。

# Write-once一致性协议状态转移表

Command	Current state	Next state	Status	Action
P-Read	无效	有效	Read-miss	读缺失时, 如远程Cache中没有重写副本, 则主存中一定有一份正确的副本, 供给发请求的Cache。如远程的Cache有重写的副本, 则它禁止主存操作, 并将副本发给请求的Cache, 两种情况均使发请求的Cache得到的副本为有效。

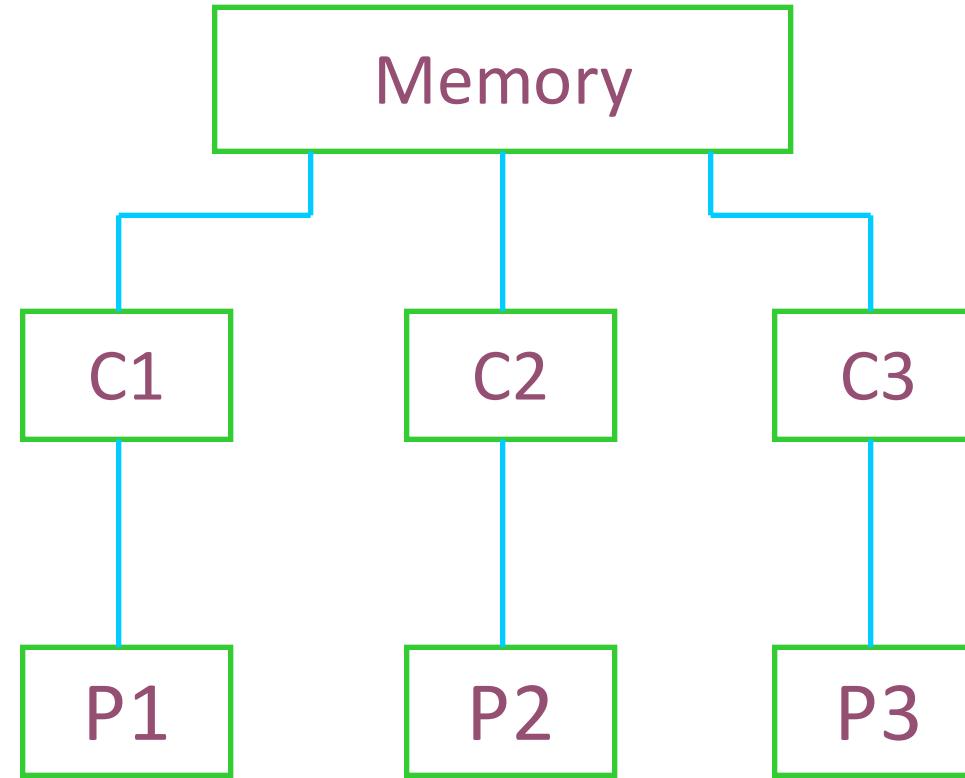
# Write-once一致性协议状态转移表

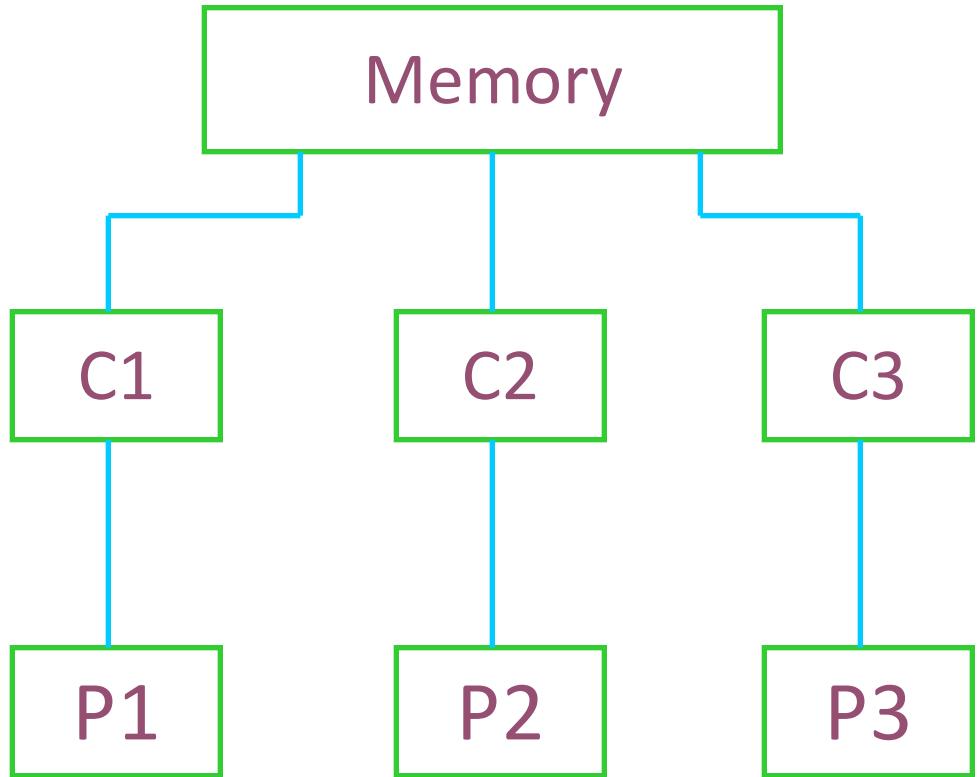
Command	Current state	Next state	Status	Action
Read-blk	保留或重写	有效	Write-hit	远程Cache读此副本，读后两份副本均有效
Read-inv	除无效外的其它状态	无效	Write-miss	写缺失时，远程Cache读一个块，并修改它，并使所有其它Cache的副本无效。

# Write-once一致性协议状态转移表

Command	Current state	Next state	Status	Action
Write-inv	有效	无效		写命中时，一远程Cache修改其本地副本，并使数据块的其它副本无效
Write-inv	有效	无效	替代	如果副本处于重写状态，必须通过块替换写回主存，否则不产生替换操作

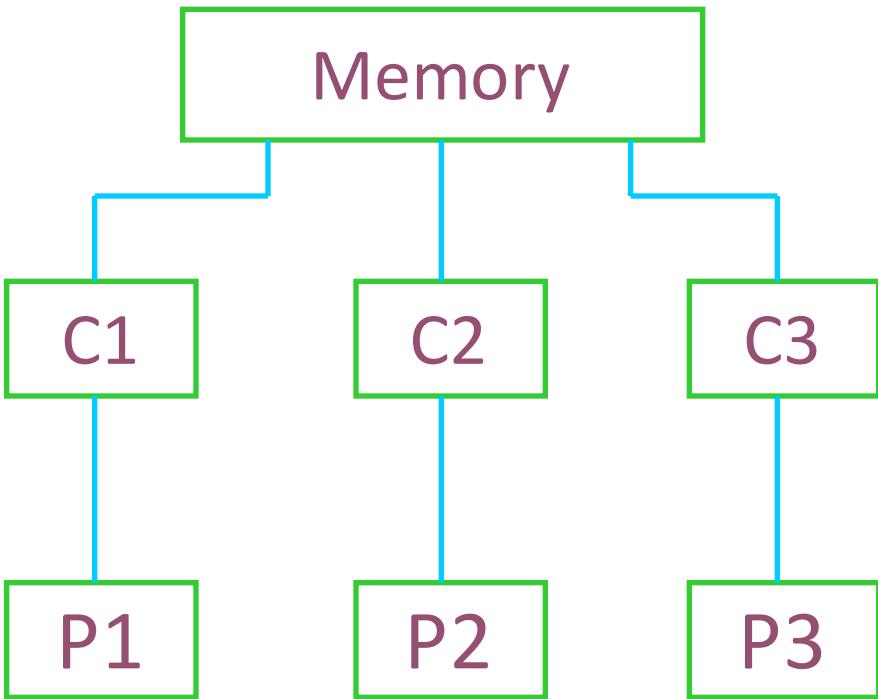
一个具体的例子





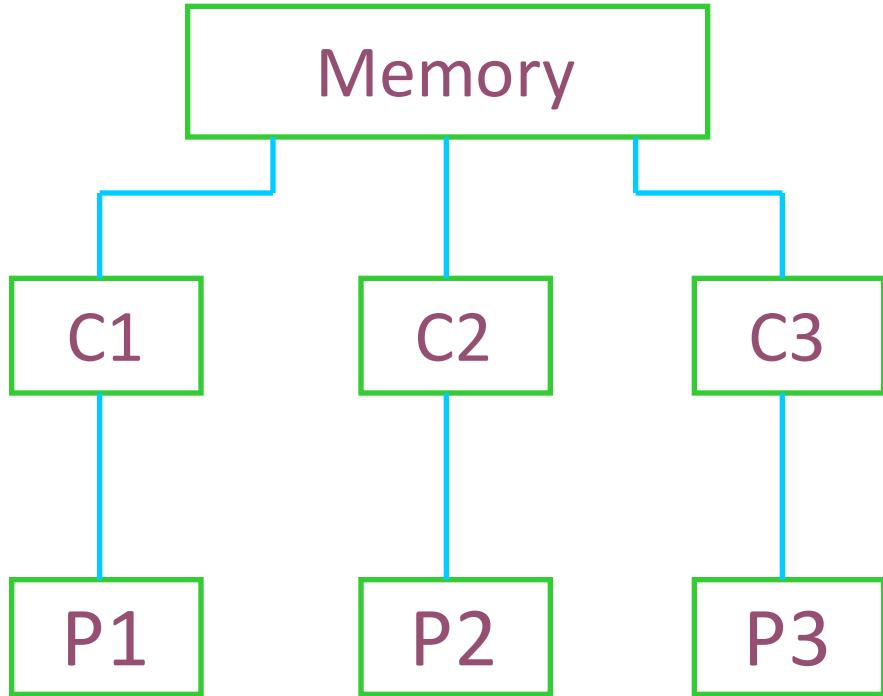
读的情况：

- 1) 如果C1为Valid， 读C1，则Read hit，状态不变。
- 2) 如果C1为Reserved， 读C1，则Read hit，状态不变。
- 3) 如果C1为Dirty， 读C1，则Read hit，状态不变。
- 4) 如果C1为Invalid， C2和C3没有东西，则读C1时Read miss，这时只有memory中有正确的副本，把它取到C1，C1改为Valid (P-Read负责实现状态的改变)。



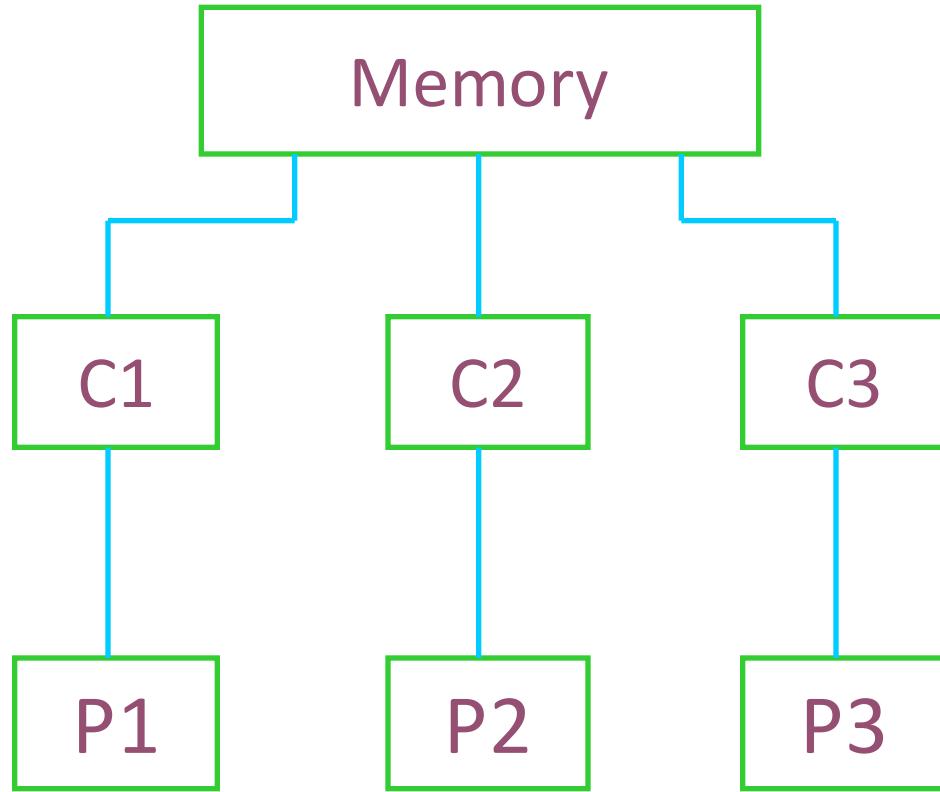
读的情况:

- 5) 如果C1为Invalid, C2为Dirty, 则读C1时Read miss, 这时只有C2中的内容是正确的, 要发Read-blk信号把副本从C2读到C1, 同时修改memory, 把C1, C2都改为Valid (程序状态转移图中 P-Read(2) 使 C1→Valid , Read-blk(3) 使C2 →Valid) 。
- 6) 如果C1为Invalid, C2为Reserved, 则读C1时 Read miss, 这时发 Read-blk 信号把 C2→C1 , C1 , C2都改为Valid, 其中 Read-blk(3) 负责把 C2 由 Reserved→Valid , P-Read(2) 负责把C1由 Invalid→Valid。

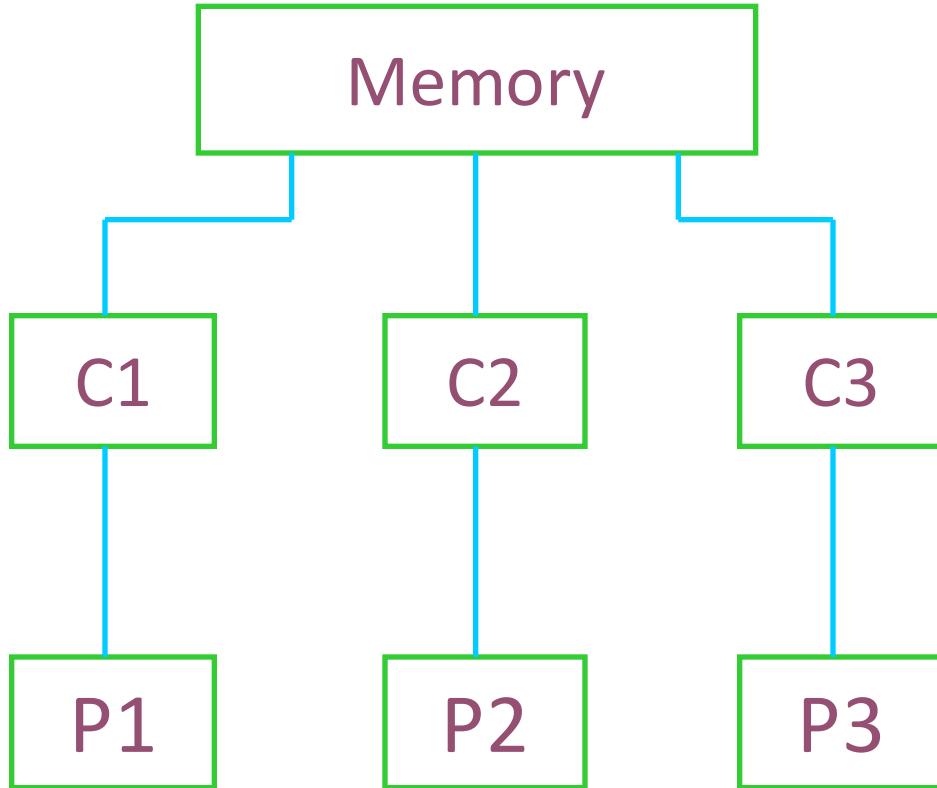


写的情况：

- 1) 如果C1为Valid，写C1，则Write hit，发P-write修改C1内容，修改memory，发Write-inv(4)给所有Cache，C1变成Reserved状态。
- 2) 如果C1为Reserved，写C1，则Write hit，发P-write修改C1内容，不修改memory，C1状态变为Dirty。
- 3) 如果C1为Dirty，写C1，则Write hit，发P-write修改C1内容，不修改memory，状态仍为Dirty。



- 4) 如果C1为Invalid， C2， C3没有东西， 这时memory中有这个地址的数据副本， 从memory中读取该副本到C1， 再把要写的内容写入C1， 这时C1和memory内容不一致， 把C1的状态变为Dirty。
- 5) 如果 C1 为 Invalid， C2 为 Dirty， 这时 memory 中内容和 C2 中的内容不一致， 把  $C2 \rightarrow C1$ ， 再把要写的内容写入 C1，  $C1 \rightarrow Dirty$ ， 发 Read-inv 使其它所有 Cache 的副本变成无效状态。



- 6) 如果C1为Invalid, C2为Reserved, 这时memory中的内容和C2内容一致, 把C2→C1, 再把要写的内容写入C1, 这时C1与memory内容不一致, 使C1→Dirty, 发Read-inv使其它所有Cache的副本变成无效状态。

# 目录表法

- 当处理机的数量很多时，一般采用多级互连网络，多级互连网络实现广播功能代价很大
- 能不能只发送给存放该副本的Cache?——基于目录的协议的基本思想

# 目录表法

- 目录里放什么——有关Cache副本驻留在哪里的信息：所有共享数据块的所有Cache副本的地址表。
- 用标志为分别指示某信息块的副本在其他几个处理机的Cache中是否存在，并记录是否已有Cache向这个信息块写入过
- 一个处理机在写入自身Cache的同时，只需要有选择的通知所有其他存在此数据块的Cache将此副本作废或更新

# 目录表的具体方式

- 全映像目录表
  - 每项有N个标志位对应于多处理机中全部N台处理机的Cache，系统中全部Cache均可存放同一个信息块的副本
  - 缺点：目录表庞大，硬件和控制均较为复杂
- 有限目录表
  - 限制N的大小，即限制了一个数据块在各Cache中能存放的副本数量
  - 以上两种方式中，目录表都集中地存放在共享主存中，因此需要由主存向各个处理机广播
- 链式目录表
  - 把目录表分散存放在各个Cache中，主存只存放一个指针，指向一台处理机
  - 要查找所有放有同一个信息块的Cache，可以先找到一台处理机的Cache，然后顺链逐台查找，直到找到目录表中的指针为空为止

# 以软件为基础实现多Cache的一致性

- 依靠软件的方法，不把一些公用可写数据存入Cache中
  - 例如，编译时，通过编译程序的分析，让属于本处理机进程私用的指令和操作数以及各处理机公用的只读型指令和数据存入Cache，而对于共享的可写数据则不让其存入Cache
- 减少硬件的复杂性，降低对互连网络通信量的要求，因而性能价格比可以比较高，比较适合处理机数量多的多处理机

# 多处理器的并行性和性能

- 多处理器的并行性既存在于指令内部，也存在于指令外部，因此，必须利用算法、程序设计语言、编译、操作系统及指令、硬件等多种途径来开发

# 并行算法的定义和分类

- 并行算法是指可同时执行的多个进程的集合，各进程可相互作用、协调和并发操作
- 按运算基本对象的不同，可分为数值型和非数值型
  - 数值型：基于代数的运算，例如矩阵运算、多项式求值等
  - 非数值型：基于关系的运算，例如选择、排序、查找等
- 按并行进程间操作顺序的不同，可分为同步型、异步型和独立型
  - 同步型：并行的各个进程需要顺序等待
  - 异步型：并行的各个进程在执行时相互独立，不会因相关而等待，根据执行情况决定终止或继续
  - 独立型：并行的各个进程间完全独立，进程之间也不需要通信

# 并行算法的定义和分类

- 根据各处理机计算任务的大小（即任务粒度）的不同，分为细粒度、中粒度和粗粒度
  - 细粒度：向量或循环级的并行
  - 中粒度：较大的循环级并行，并确保这种并行带来的性能的提升可以补偿其开销
  - 粗粒度：子任务级的并行

# 多处理机并行的研究思路

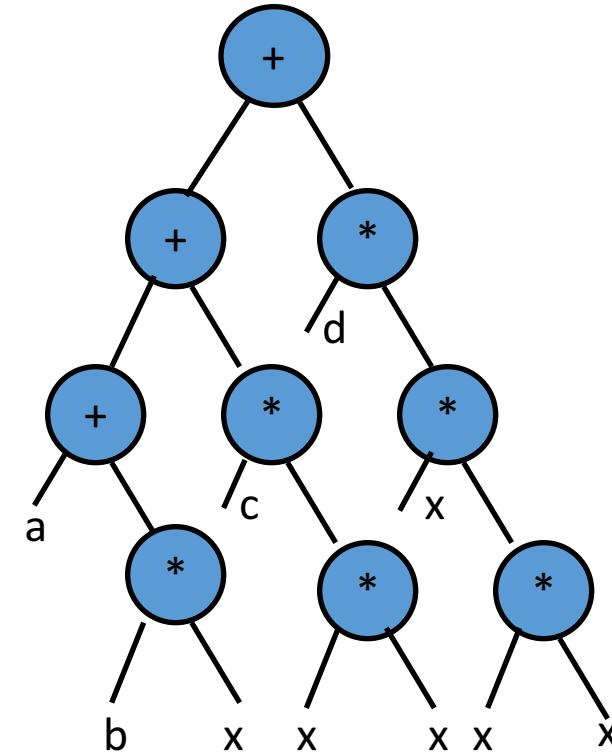
- 将大的程序分解成足够多的可并行处理的过程（进程、任务、程序段等）
- 每个过程看做一个结点，则过程之间的关系可以用这些结点组成的树来表示
- 这种树可以理解成各个子过程输出结果之间的运算，即算数表达式中各项之间的运算

# 性能评价

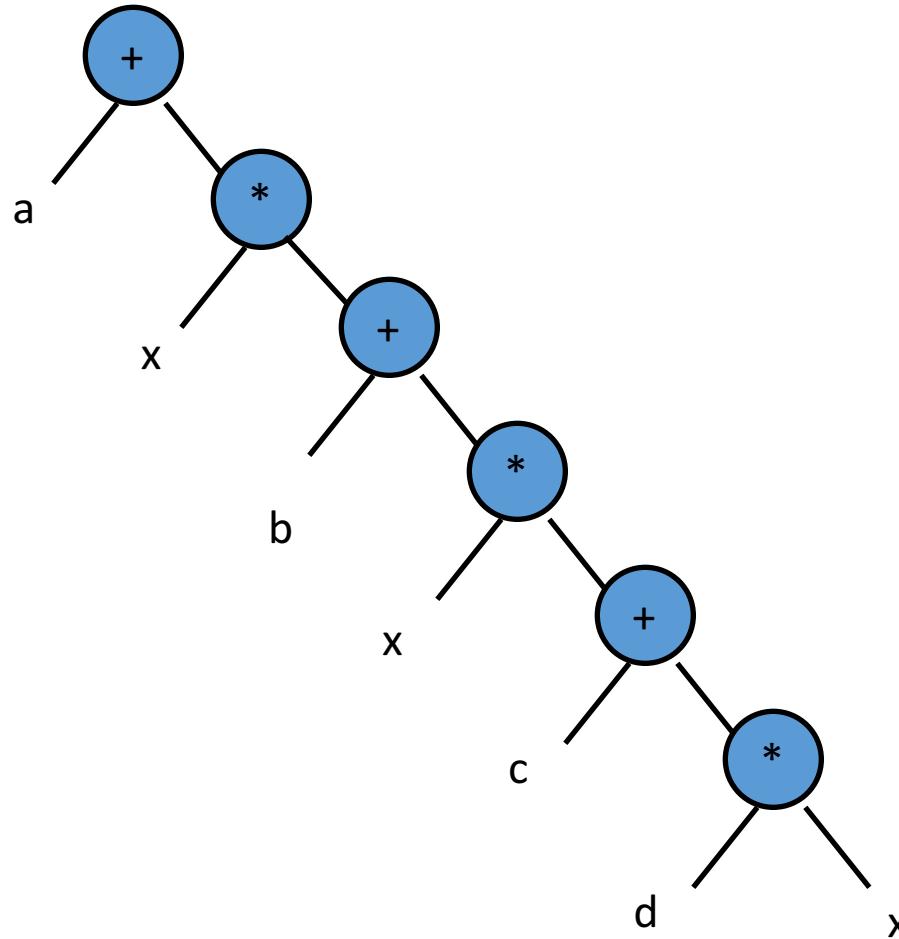
- $P$ 表示处理机的数量
- $T_P$ 表示 $P$ 台处理机运算的级数，即树的高度
- $S_P = T_1/T_P$ 表示多处理机并行运算相对于单处理机顺序运算的加速比
- $E_P = S_P/P$ 表示 $P$ 台处理机的设备利用率

# 举例

- $E_1 = a + bx + cx^2 + dx^3$
- 用3台处理机，需4级运算
- 级数（高度）  $T_p = 4$
- 处理机数  $P = 3$
- 加速比  $S_p = \text{顺序运算级数}/\text{高度} = 6/4 = 3/2$
- 效率  $E_p = S_p/P = 1/2$



- $E_1 = a+bx+cx^2+dx^3$   
 $= a + x(b+x(c+x(d)))$
- 单处理机需要6级运算

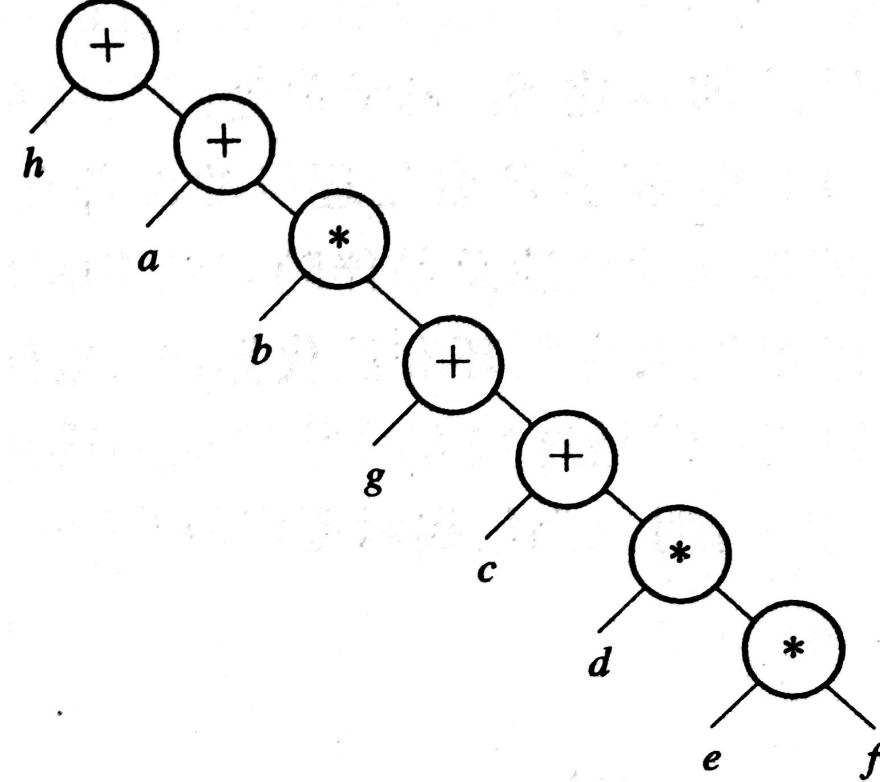




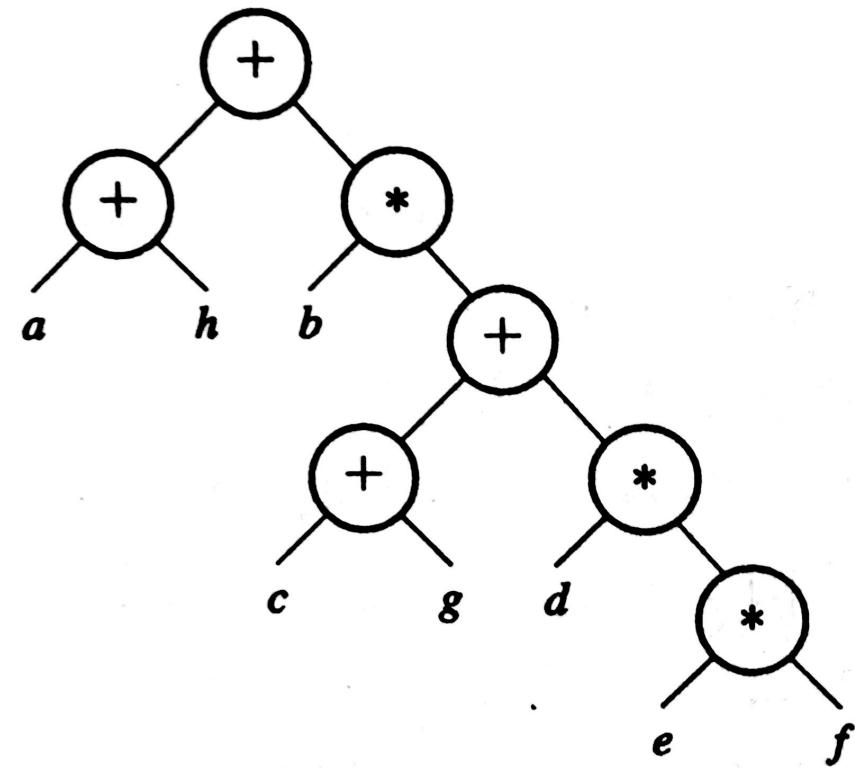
- 减少运算的级数，降低树高，增大树的广度
- 用交换律、结合律、分配律来变换
- 先利用交换律把相同的运算集中起来，再用结合律把参加运算的操作数配对，尽可能并行运算，最后再把子树结合起来。

- 计算  $E_2 = a + b(c + def + g) + h$
- 单处理机情况下

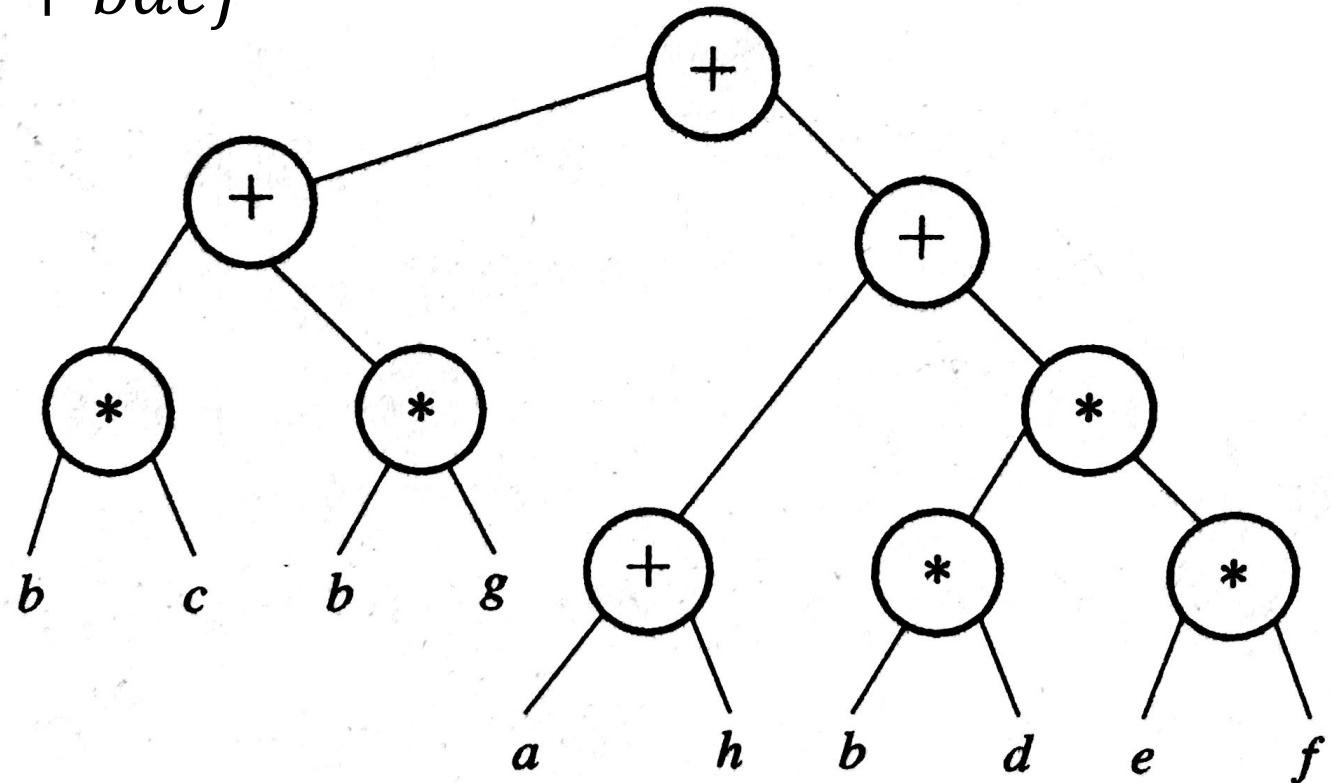
- $P = 1$
- $T_p = 7$



- $E_2 = (a + h) + b((c + g) + def)$
- $P = 2$
- $T_P = 5$
- $S_P = 7/5$
- $E_P = 0.7$



- $E_2 = (a + h) + (bc + bg) + bdef$
- $P = 2$
- $T_P = 5$
- $S_P = 7/5$
- $E_P = 0.7$



# 程序的并行性分析

- 程序段 $P_1, P_2, \dots, P_i, \dots, P_j, \dots, P_n$

程序段 $P_1, P_2, \dots, P_i, \dots, P_j, \dots, P_n$

# 程序的并行性分析

- 程序段  $P_1, P_2, \dots, P_i, \dots, P_j, \dots, P_n$
- $P_j$  “数据相关” 于  $P_i$ :  $P_i$  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

$$P_i: A = B + D$$

$$P_j: C = A * E$$

虽无法并行，但如果能够交换串行，则可以让空闲处理机先执行  $P_j$ ，有利于从宏观上提程序段之间的并行

- 特例:  $P_i$  和  $P_j$  服从交换律，则可以交换串行

$$P_i: A = 2 * A$$

$$P_j: A = 3 * A$$

- 两个程序段之间存在先写后读的数据相关，则不能并行；只有在特殊情况下可以交换串行

- $P_j$  “数据反相关”于  $P_i$ :  $P_j$  的左部变量在  $P_i$  的右部变量集内，且  $P_i$  未取用其变量的值之前

$$P_i: C = A + E$$

$$P_j: A = B + D$$

- 当  $P_i$  和  $P_j$  并行时，若保证  $P_i$  对  $A$  先读出，就能得到正确的结果
  - 让每个处理机的操作结果先暂存于自己的局部存储器（或 Cache 存储器）中，不急于修改原来存放在共享主存单元的内容
- 若有先读后写数据反相关，则可以并行，单必须保证其写入共享主存时的先读后写次序，不能交换串行

- $P_j$  “数据输出相关” 于  $P_i$ :  $P_j$  的左部变量也是  $P_i$  的左部变量，且  $P_j$  存入其结果必须在  $P_i$  存入之后

$$P_i: A = B + D$$

$$P_j: A = C + E$$

- 若有写—写的数据出处相关，可以并行执行，但必须保证其写入的先后次序，不能交换串行

- 以交换数据为目的的程序段同时具有“先写后读”和“先读后写”两种相关

$$P_i: A = B$$

$$P_j: B = A$$

- 若同时具有“先写后读”和“先读后写”两种相关，以交换数据为目的时，必须并行执行，且读、写要完全同步，不能交换串行或顺序串行

# 并行语言和并行编译

- 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**: 一个任务在执行的同时，可以派生出可与它并行执行的其他一个或多个任务，分配给不同的处理机完成
- 汇合**JOIN**: 派生出的任务完成后，再汇合起来进行后续的单任务或新的并行任务
- **FORK**和**JOIN**在不同机器上有不同的表现形式

# M.E.Conway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
- JOIN n:
  - 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。

- 计算

$$Z = E + A * B * C / D + F$$

- 并行编译:

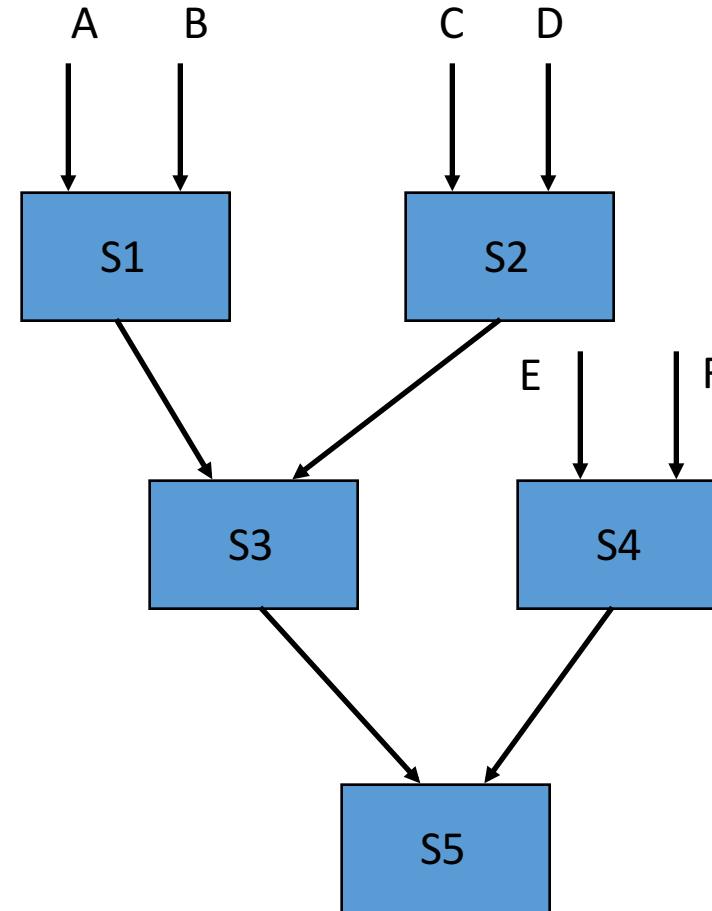
$$S1 \quad G = A * B$$

$$S2 \quad H = C / D$$

$$S3 \quad I = G * H$$

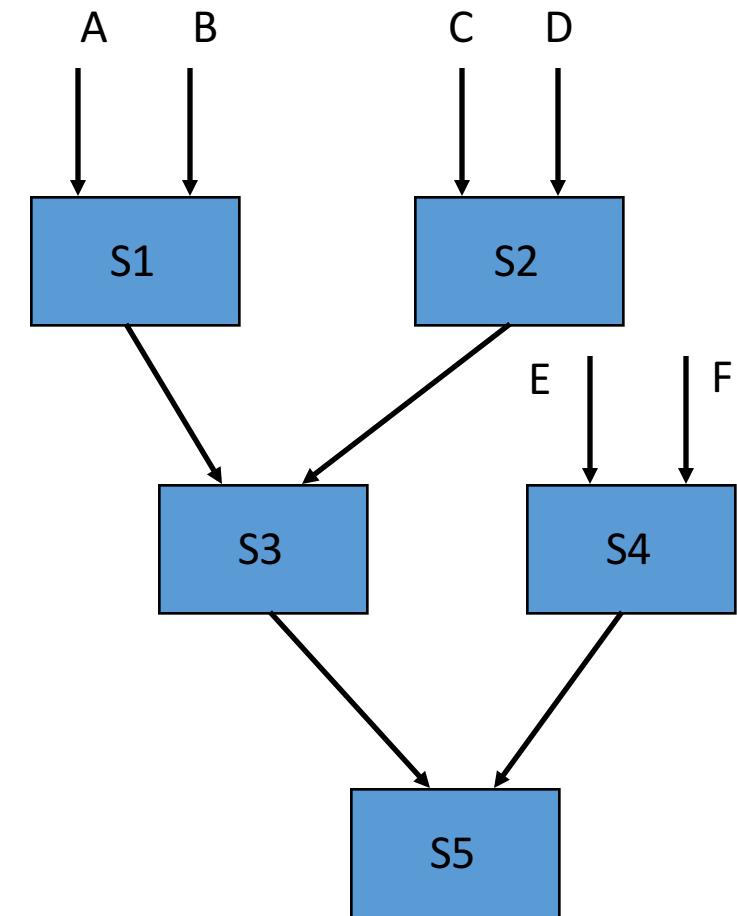
$$S4 \quad J = E + F$$

$$S5 \quad Z = I + J$$

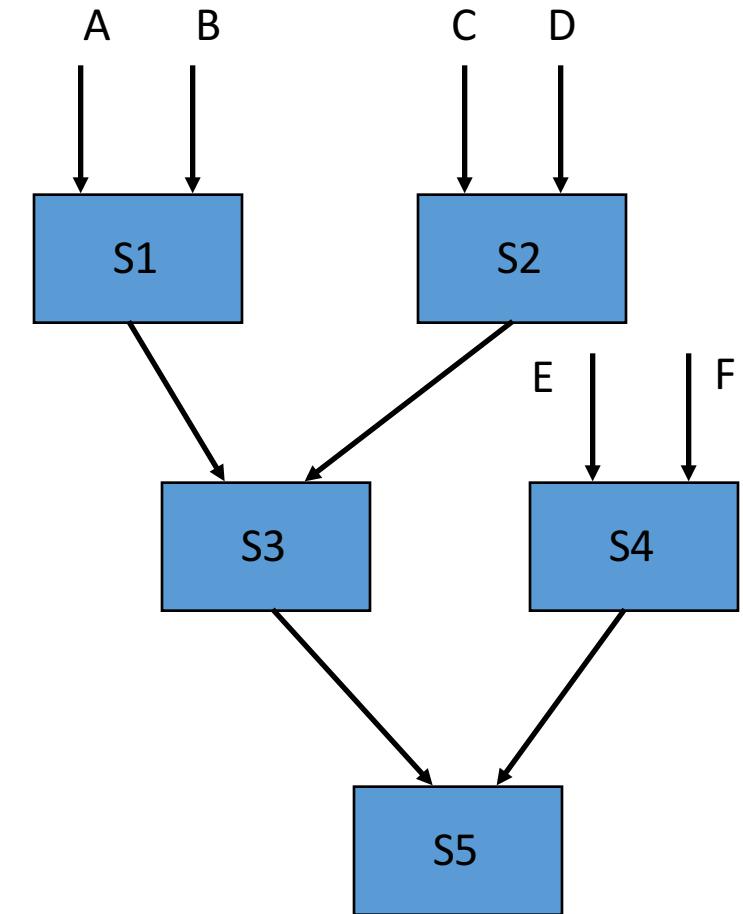
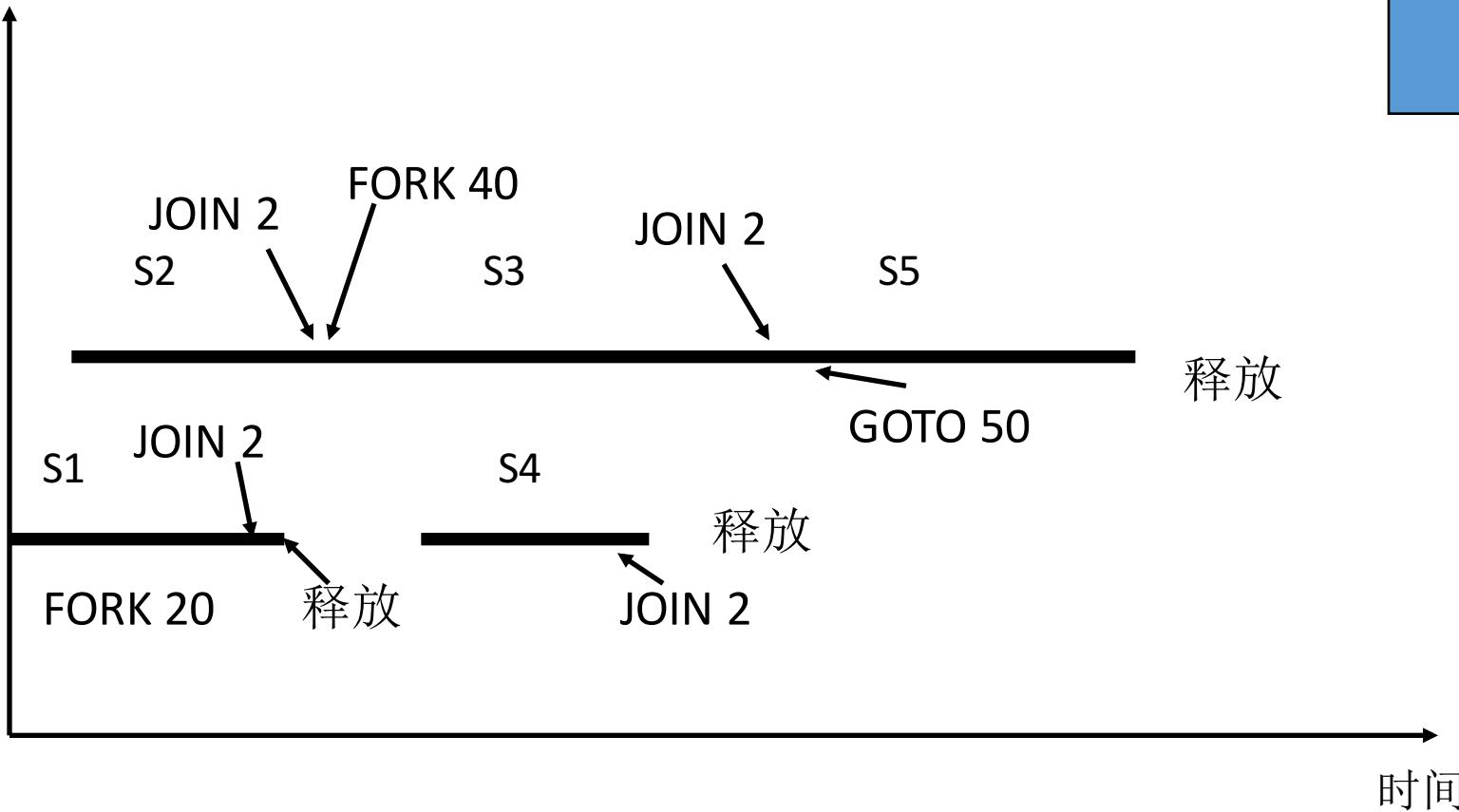


	FORK 20	
10	G=A*B (进程S1)	
	JOIN 2	
	GOTO 30	
20	H=C/D (进程S2)	
	JOIN 2	

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)	



处理机

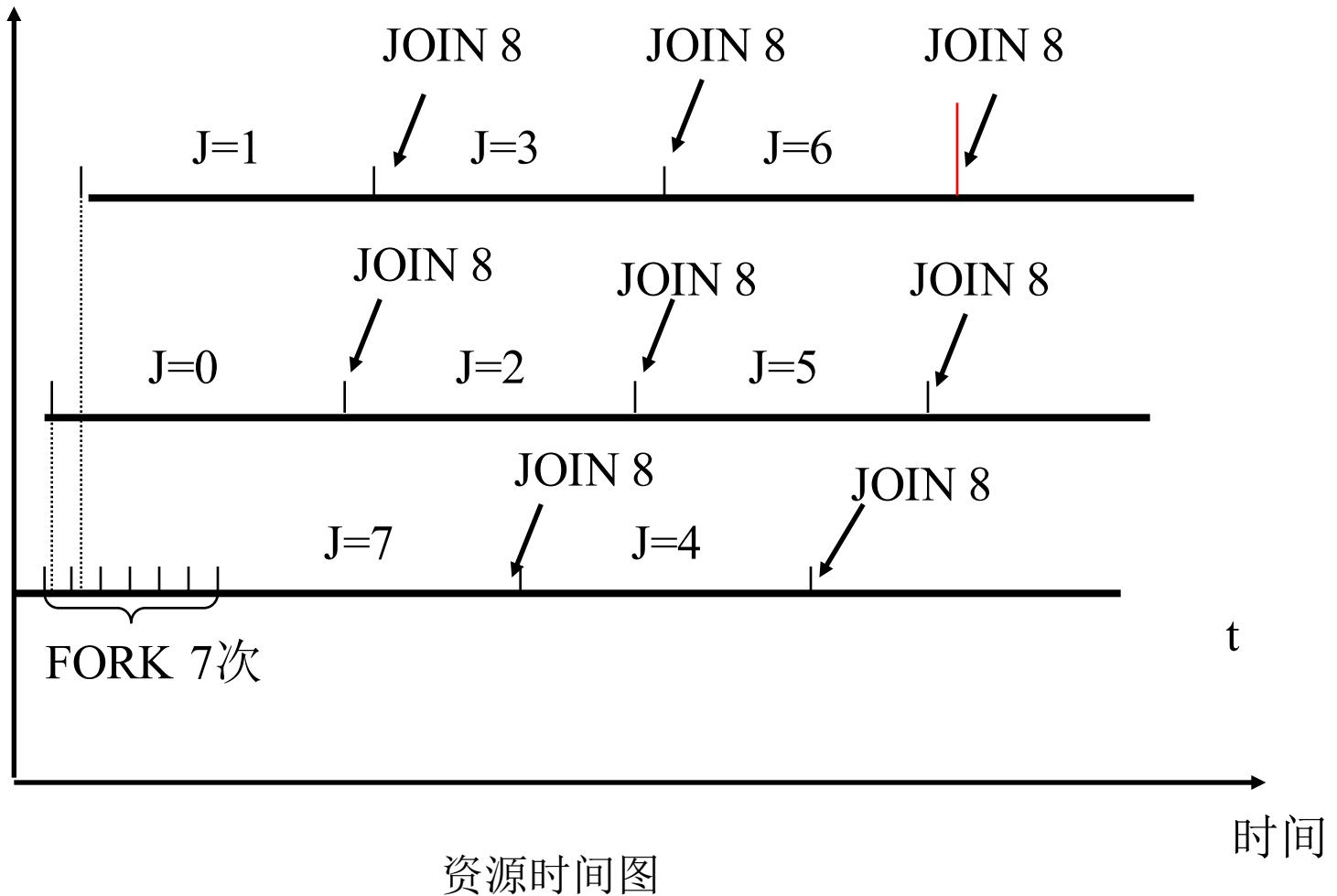


## 举例2

- 假定A，B两个8\*8矩阵相乘：

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

处理机



# 多处理机与并行处理机的区别

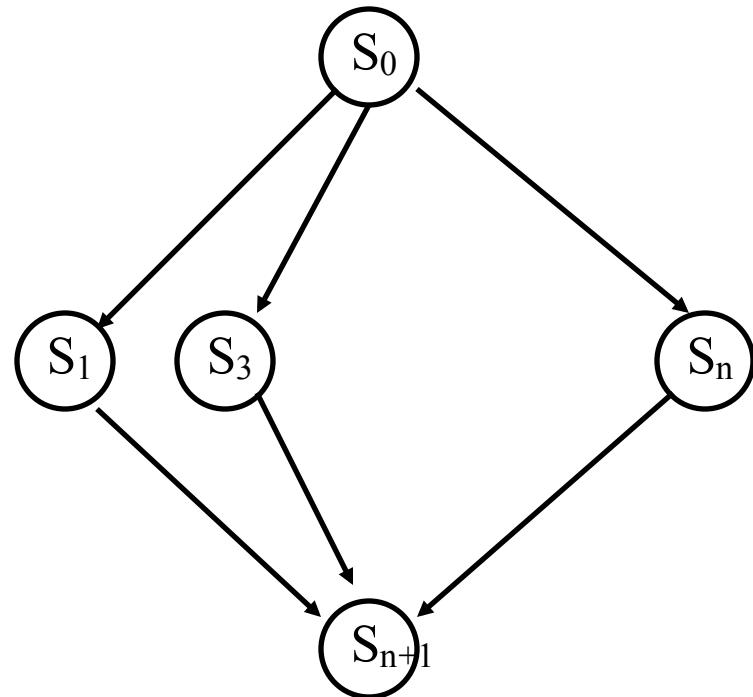
- 并行处理机的每一条指令要求8个处理单元完全同步地对 $j=0,1,\dots,7$ 的不同数组进行运算。在多处理机中，不需要也不会完全同步。
  - 操作级与任务级
- 多处理机中可用处理器数目对程序编写没有影响。

# 块结构语言

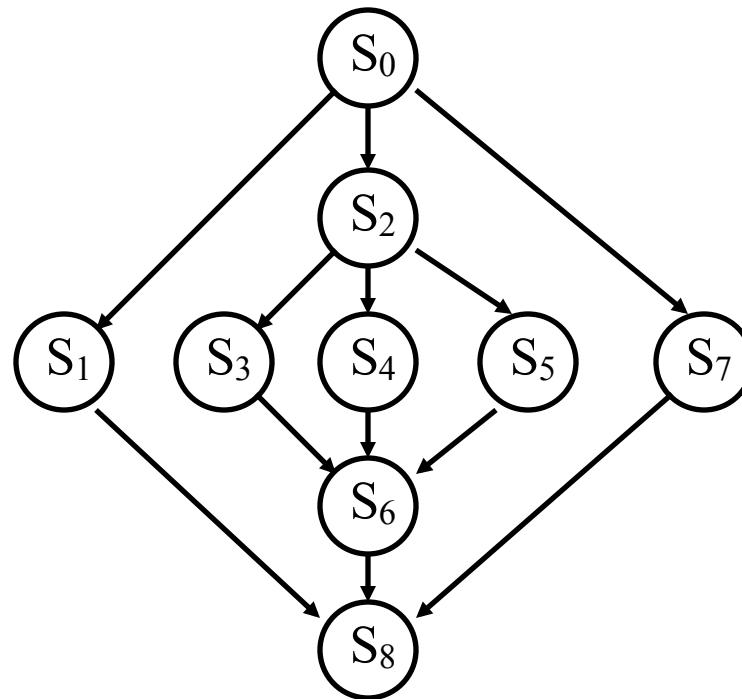
- E.W.Dijkstra提出
- 把可并行执行的进程用cobegin-coend（或parbegin-parend）括起来处理,最后一条语句执行完成后，方可执行后续语句。
- 该语句可嵌套；可使用共享变量，但不允许修改。

# 举例1

```
begin  
S0;  
Cobegin S1;S2,⋯;Sn;coend  
Sn+1;  
end
```



```
Begin
    S0;
    cobegin
        S1;
        begin
            S2;
            cobegin S3;S4;S5;coend
            S6;
        end
        S7;
    coend
    S8;
end
```



# 多处理器性能

- 引起峰值性能下降的原因是：
  - 因处理机间通信而产生的延迟
  - 一台处理机与其它处理机同步所需的开销
  - 当没有足够多任务时，一台或多台处理机处于空闲状态
  - 由于一台或多台处理机执行无用的工作
  - 系统控制和操作调度所需开销

# 多处理器性能（续）

- 研究多处理器的目的：
  - 提前5年得到速度高10倍的机器。或用 $1/10$ 的价格获得一台高性能的机器。
  - 如果设计得好，在某些适合进行并行处理得应用领域，可以达到：提前10年得到速度高100倍的机器 或用 $1/100$ 的价格获得一台高性能的机器。
  - 并行性在很大程度上依赖于E/C比值， 其中：
    - E代表程序执行时间
    - C代表通信开销。

# 多处理器性能（续）

- 通常： $E/C$ 比值小，并行性低。 $E/C$ 比值大，并行性高
- 如果把作业分解成较大的块，就能得到较大的 $E/C$ 值，但是所得到的并行性比最大可能的并行性要小得多。
- $E/C$ 比值是衡量任务粒度(Granularity)大小的尺度
- 在粗粒度（Coarsegrain）并行情况下， $E/C$ 比值比较大，通信开销小

# 多处理器性能（续）

- 在细粒度并行情况下， $E/C$ 比值比较小，通信开销大
- 细粒度并行性需要的处理机多，粗粒度并行性需要的处理机少。
- 细粒度并行性的基本原理是把一个程序尽可能地分解成能并行执行的小任务。在极端情况下，一个小任务只完成一个操作。

# 多处理器的操作系统

- 多处理器操作系统的难度与特点
- 多处理器操作系统的类型
- 多处理器操作系统的的发展

# 多处理器操作系统的难度

- 处理机的分配和进程调度
- 进程间的同步
- 进程间的通信
- 存储系统的管理
- 文件系统的管理
- 系统重组

# 多处理器操作系统的概念

- 程序执行的并行性
- 分布性
- 机间通信与同步性
- 系统容错性

# 多处理器操作系统的类型

- 主从型(Master-slave Configuration)
  - 管理程序只在主处理机上运行。
  - 硬件结构、管理、控制简单，对主处理机要求高。
  - 用于工作负荷固定、从处理机能力明显低的紧耦合、异构型、非对称多处理机系统。
  - 实现简单，经济，方便

# 多处理器操作系统的类型（续）

- 各自独立型（Separate Supervisor）
  - 每个处理机有独立的管理程序在运行。
  - 管理程序可再入，可靠性高，系统表格少，系统效率高；实现复杂、访存冲突解决和负载平衡较困难。
  - 效率高；
  - 实现复杂，
  - 适合于松耦合多处理机

# 多处理器操作系统的类型（续）

- 浮动型(Floating Supervisor)
  - 管理程序在多个处理机间浮动。
  - 管理程序可再入，实现复杂，负载平衡较好。
  - 折衷方式；
  - 灵活性；
  - 适合于紧耦合型，同构型；