计算机体系结构

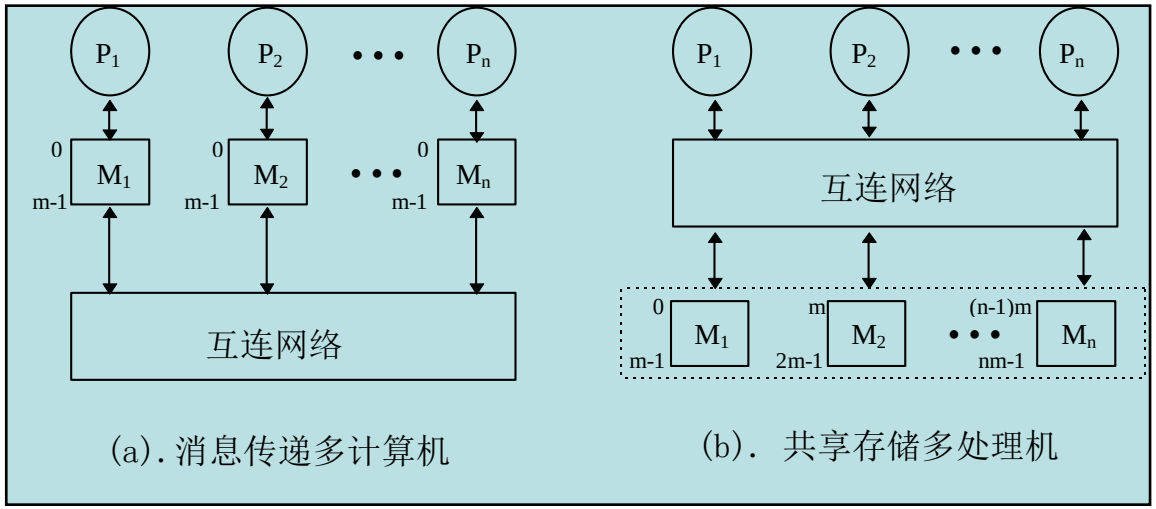
胡伟武、汪文祥

多处理器

- 消息传递与共享存储
- 共享存储系统中的访存相关性
- 共享存储系统的访存事件次序
- 存储一致性模型
- CACHE一致性协议

消息传递与共享存储

两种并行系统：消息传递与共享存储



- 多地址空间
- 消息传递通讯
- 编程困难、程序移植困难
- 通用性差
- 可伸缩性好

- 单地址空间
- 共享存储通讯
- 编程容易、程序易移植
- 通用性强
- 可伸缩性一般

共享存储与消息传递的编程复杂度

- 任务划分与数据划分
  - 共享存储编程只需划分任务
  - 消息传递编程除了划分任务外，还需划分数据和考虑通信
  - 微信与短信
- 传递复杂的数据结构较困难
  - 多个指针组成的结构
  - struct {int *pa; int pb*; int *pc}
- 动态通信
  - {for (i,j){ x=…; y=…; a[i][j]=b[x][y];}}
  - 进程迁移及进程数目的变化

例子：积分求π

$$\pi = 4\int_0^1 \frac{1}{1+x^2}dx = \sum_{i=1}^{N} \frac{4}{1+\left(\frac{i-0.5}{N}\right)^2} \times \frac{1}{N}$$

串行程序

```
h = 1.0/N; pi = 0;
for(i=1; i<=N; i++){
      temp = (i-0.5)*h;
      pi = pi + 4/(1+temp*temp);
}
pi = pi * h;
printf pi;
```

积分求π的并行程序

JIAJIA

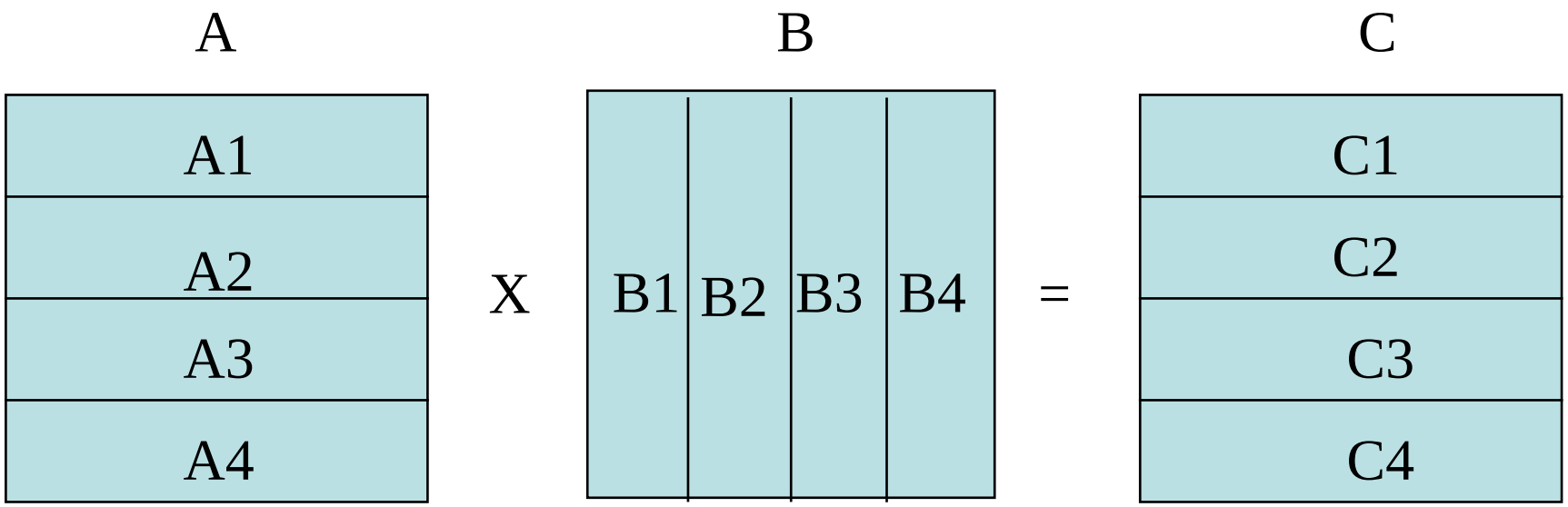
```
jia_init(argc,argv);
pi=jia_alloc(8);

h = 1.0/N;
for(i=jiapid+1;i<=N;i+=jiahosts)
   { mypi = ... }
jia_lock(1);
   pi += mypi;
jia_unlock(1);
printf pi;
jia_exit();
```

MPI

```
MPI_Init(argc,argv);
MPI_Comm_size();
MPI_Comm_rand();
h = 1.0/N;
for(i=myid+1;i<=N;i+=numprocs)
   { mypi = ... }
MPI_Reduce(mypi,pi,1   ... );
printf pi;
MPI_Finalize();
```

并行程序例子：矩阵乘法



- A × B = C的过程可分为四个独立的部分：

  Ai × B = Ci，i = 1，2，3，4

- 每部分包含的运算可由一台处理机单独完成
- 矩阵较大时，需分开存放，即Ai,Bi,Ci放在结点i上
- 每个结点的工作分为AiB1,AiB2,AiB3,AiB4四个部分

矩阵乘法的并行程序

- 共享存储

```
double (*a)[N],(*b)[N],(*c)[N];
a=jia_alloc(N*N*8);
b=jia_alloc(...); c=jia_alloc(...);
if (jiapid==0) for (i...) for (j…){
  a[i][j]=1;b[i][j]=1;
}
jia_barrier();
begin=N*jiapid/jiahosts;
end=N*(jiapid+1)/jiahosts;
for (i=begin; i<end; i++)
    for (j=0; j<N; j++)
          for (k=0; k<N; k++)
            c[i][j]+=a[i][k]*b[k][j];
jia_barrier();
if (jiapid==0) printf C;
jia_exit();
```

- 消息传递

```
double (*a)[N],(*b)[N],(*c)[N];
a=malloc2(N*N*8);
b=malloc2(...); c=malloc(...);
if (mypid==0) {
  init a,b; send a,b;
}else{
  recv a,b;
}

for (i=0;i<N/hosts;i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      c[i][j]+=a[i][k]*b[k][j];

if (mypid!=0){  send c;}
else{ recv c; printf c;}
```

比较：Pthreads并行程序举例-矩阵乘法

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4 //假设线程数目为4
#define n 1000
double *A,*B,*C;
void *matrixMult(void *id) {//计算矩阵乘
    int my_id = (int ) id;
   int i,j,k,start,end;
//计算进程负责的部分
  start = my_id*(n/NUM_THREADS);
  if(my_id == NUMTHREADS-1)
      end = n;
  else
     end = start+(n/NUM_THREADS);
  for(i=start;i<end;i++)
        for(j=0;j<n;j++) {
            C[i*n+j] = 0;
           for(k=0;k<n;k++)
                C[i*n+j]+=A[i*n+k]*B[k*n+j];
         }
 }
```

```
int main() {
  int i,j;
  pthread_t tids[NUM_THREADS];

  //分配数据空间
  A = (double *)malloc(sizeof(double)*n*n);
  B = (double *)malloc(sizeof(double)*n*n);
  C = (double *)malloc(sizeof(double)*n*n);

    //初始化数组
  for(i=0;i<n;i++)
     for(j=0;j<n;j++){
        A[i*n+j] = 1.0;
        B[i*n+j] =  1.0;
      }

  for(i=0; i<NUM_THREADS; i++)
        pthread_create(&tids[i], NULL,
matrixMult, (void *) i);
 for(i=0; i<NUM_THREADS; i++)
          pthread_join(tids[i], NULL);
   return 0;
}
```

比较：OpenMP并行矩阵乘法

```
#include <stdio.h>
#include <omp.h>
#define n 1000
double A[n][n],B[n][n],C[n][n];
int main()
{
    int i,j,k;
    //初始化矩阵A和矩阵B
    for(i=0;i<n;i++)
    for(j=0;j<n;j++) {
      A[i][j] = 1.0;
      B[i][j] = 1.0;    }

    //并行计算矩阵C
    #pragma omp parallel for shared(A,B,C) private(i,j,k)
    for(i=0;i<n;i++)
      for(j=0;j<n;j++){
        C[i][j] = 0;
        for(k=0;k<n;k++)
         C[i][j]+=A[i][k]*B[k][j];}
   Return 0;
}
```

MPI程序例子-矩阵乘法

```
#include <stdio.h>
#include “mpi.h”
#define n 1000
int main(int argc, char **argv) {
   double *A,*B,*C;
   inti,j,k, ID,num_procs,line;
   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,
            &ID);//获取当前进程号
   MPI_Comm_size(MPI_COMM_WORLD,
            &num_procs);//获取进程数目
  //分配数据空间
  A = (double *)malloc(sizeof(double)*n*n);
  B = (double *)malloc(sizeof(double)*n*n);
  C = (double *)malloc(sizeof(double)*n*n);
  line = n/num_procs;//按进程数来划分数据
  if(ID==0){ //节点0，主进程
      for(i=0;i<n;i++) //初始化数组
      for(j=0;j<n;j++){
         A[i*n+j] = 1.0;B[i*n+j] =  1;}
    //将矩阵A、B的相应数据发送给从进程
     for(i=1;i<num_procs;i++)  {
      MPI_Send(B,n*n,  MPI_DOUBLE,
           i,0,MPI_COMM_WORLD);
      MPI_Send(A+(i-1)*line*n,line*n,
      MPI_DOUBLE,i,1,MPI_COMM_WORLD);
    }
  //接收从进程计算结果
  for(i=1;i<num_procs;i++)
       MPI_Recv(C+(i-1)*line*n,line*n,
            MPI_DOUBLE,i,2,MPI_COMM_WORLD,&Status);
  //计算剩下的数据
  for(i=(num_procs-1)*line;i<n;i++)
     for(j=0;j<n;j++) {
       C[i*n+j]=0;
       for(k=0;k<n;k++)
          C[i*n+j]+=A[i*n+k]*B[k*n+j]; }
  } else {  //其他进程接收数据，计算结果，发送给主进程
      MPI_Recv(B,n*n,MPI_DOUBLE,0,0,
           MPI_COMM_WORLD,&Status);
      MPI_Recv(A+(ID-1)*line*n,line*n,
           MPI_DOUBLE,0,1,MPI_COMM_WORLD,&Status);
     for(i=(ID-1)*line;i<ID*line;i++)
        for(j=0;j<n;j++) {
          C[i*n+j]=0;
          for(k=0;k<n;k++)
            C[i*n+j]+=A[i*n+k]*B[k*n+j];
        }
     MPI_Send(C+(num_procs-1)*line*n,line*n,
           MPI_DOUBLE,0,2,MPI_COMM_WORLD);
  }
  MPI_Finalize();
  Return 0;
} //main
```

并行程序的例子：图象纠正

- 串行程序

```
for (i=0;i<lin2;i++){
  for (j=0;j<col2;j++){
    x=P(i,j,…);
    y=P(i,j,…);
    out[i,j]=in[x][y];
  }
}
```

- 共享存储并行程序

```
start=lin2/jiahosts*jiapid;
end=start+lin2/jiahosts;
if (jiapid==jiahosts) end=lin2;
for (i=start;i<end;i++){
  for (j=0;j<col2;j++){
    x=P(i,j,…);
    y=P(i,j,…);
    out[i,j]=in[x][y];
  }
}
jia_barrier();
```

常见的并行处理结构

- SIMD结构
  - 早期Cray向量机，现代CPU的SIMD短向量指令（128、256、512位）
- SMP结构：多核共享存储
  - 早期多路服务器/工作站（DEC、SUN、SGI），现在片内多核
- CC-NUMA结构：更多核共享存储
  - SGI、IBM、HP高端服务器，几十到上千路共享内存服务器
  - 片内核数增加导致CC-NUMA
- MPP或机群结构
  - HPC，如曙光高性能机、太湖之光，主要用于科学计算
  - 云计算，机群数据库
- GPU（上千核）采用什么结构？

共享存储与消息传递发展趋势

- 1970年代到1980年代中期，并行系统主要是SMP与向量机，结点个数不多（4-8路），是科学计算、事务处理、服务器领域的主要产品
- 由于SMP与向量机可伸缩性差，1990年代消息传递的MPP兴起，主要用于科学计算，同时NUMA系统的研究全面展开，如DASH，Alewife，KSR-1等
- 1990年代后期，由于解决了伸缩性问题，以Origin 2000为标志，中规模和大规模的共享存储兴起，SUN的Starfile，Compaq的Wildfire，及IBM的NUMA-Q
  - 目前主流的服务器均采用共享存储，低端服务器由处理器提供直接互连形成2-4路系统；高端服务器通过桥片连接形成几十路的系统
- 以科学计算为目的大规模MPP系统仍以消息传递为主，几千到上万个处理机，随着带宽的增加与用户级通信的发展，机群系统与MPP系统的界限越来越模糊
- 主流商业片内多核都采用共享存储结构，甚至分布式共享存储系统；片内众核（几百到上千核）如GPU结构采用独立存储结构

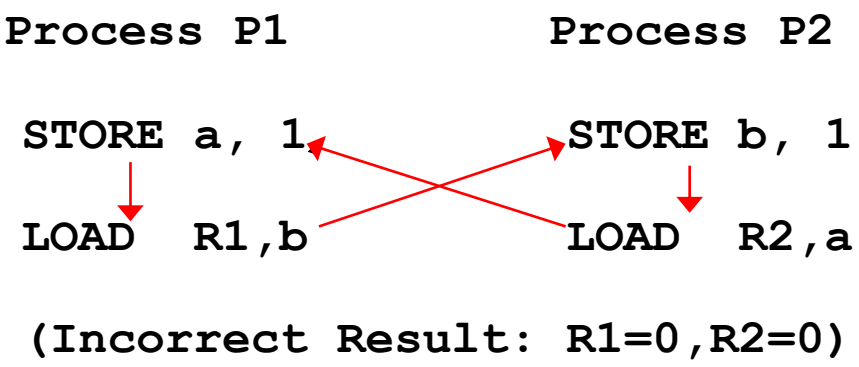
共享存储多核处理器的关键问题

- 通用多核处理器一般采用共享存储结构
- 多个处理器核发出的访存指令次序如何约定？
  - 存储一致性模型：如顺序一致性、处理器一致性等
- 多个处理器核间共享片上Cache如何组织及维护一致性？
  - Cache一致性协议：片上Cache结构及Cache一致性协议
- 多个核处理器核间如何实现通信？
  - 片上互连结构
- 多个处理器核间如何实现同步？
  - 多核同步机制：互斥锁操作（lock）、路障操作（barrier）

共享存储系统中的访存相关性

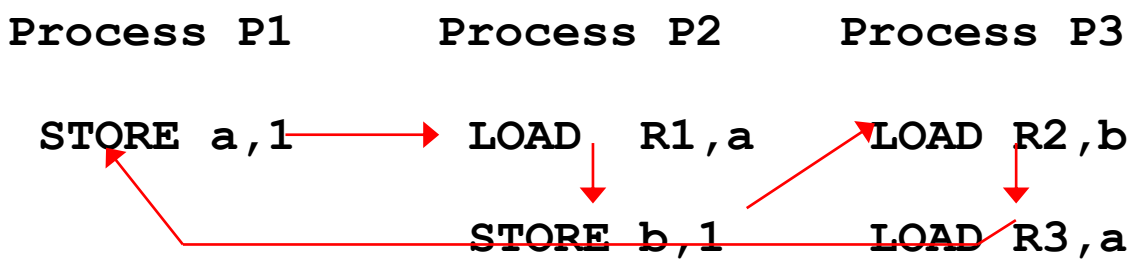
一致性问题

- 在单机系统中，只要保持程序中的数据相关性，就可以保证执行正确。在多处理机系统中，不仅要考虑单机内的数据相关，而且要考虑多机之间的数据相关
- 为了执行的正确性，每个处理机都必须根据程序序来执行指令



一致性问题（cont.）

- 即使每个处理机都根据程序序执行指令，仍可能导致错误结果



(Incorrect result: R1=1,R2=1, R3=0)

- 起因：从Write Atomic到Write Non-atomic
- 什么是写可分割系统中正确的执行？
  - 直观感觉，程序次序与执行次序不要形成闭环
  - 存储一致性模型和Cache一致性协议

什么是正确的执行？

- 正确性标准：符合程序员的直觉
- 正确性规范：顺序一致性
  - Sequential consistency defines a correct execution as the one “whose result is the same as if the operations of each individual processor appear in this sequence in the order specified by the program”.
  - 如果在多处理机环境下的一个并行执行的结果等于同一程序在单处理机多进程环境下的一个执行的结果，则此并行程序执行正确

并行程序的顺序执行

```
Process P1          Process P2

STORE a, 1          STORE b, 1

LOAD  R1,b          LOAD  R2,a
```

ST a,1	ST b,1	ST b,1	LD R2,a
LD R1,b	LD R2,a	ST a,1	LD R1,b
ST b,1	ST a,1	LD R2,a	ST b,1
LD R2,a	LD R1,b	LD R1,b	ST a,1
（R1=0，R2=1）	（R1=1，R2=0）	（R1=1，R2=1）	（R1=0，R2=0）

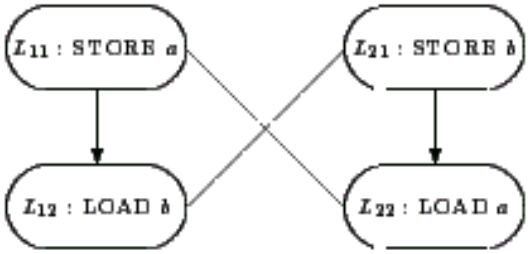
并行程序模型

- 程序模型
  - 一个进程P是一个二元组<V(P),PO(P)>，其中V(P)是LOAD和STORE指令的集合，PO(P)是V(P)上的一个全序关系
  - 由N个进程P1,P2,…,Pn组成的程序PRG(P1,P2,…,Pn)是一个二元组<V(PRG),PO(PRG)>，其中V(PRG)=V(P1)∪ V(P2)∪...∪V(Pn)是程序PRG的指令集，PO(PRG)=PO(P1)∪PO(P2)∪...∪PO(Pn)是程序PRG的程序序
- 冲突访问的概念
  - 如果两个访存操作访问的是同一单元且其中至少有一个是存数操作，则称这两个访存操作是冲突的。
  - C(PRG)={(u,v)|((u,v)∈V(PRG))∩(u,v是冲突访问)}称为程序PRG中冲突访问对集

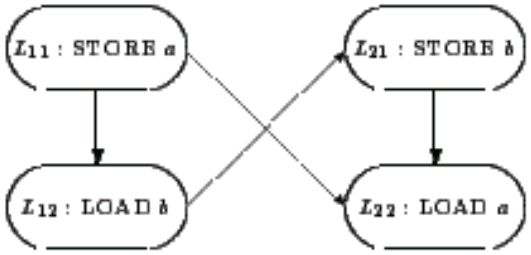
程序执行的正确性

- 执行的概念
  - 在在一个并行执行中，一旦互相冲突的访问的执行次序确定了。那么执行结果也就确定了。
  - 在程序PRG中，对冲突访问对集C(PRG)的任一无圈定序称为程序PRG的一个执行，记为E(PRG)。
- 执行的正确性
  - 程序PRG的执行E(PRG)正确的充要条件是E(PRG) ∪ PO(PRG)无圈。

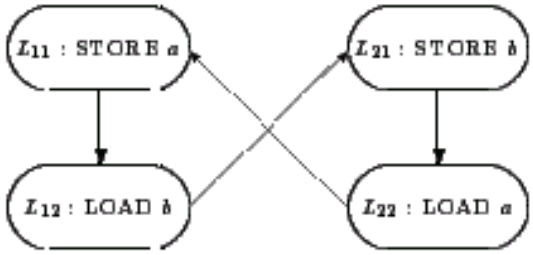
正确与错误的程序执行



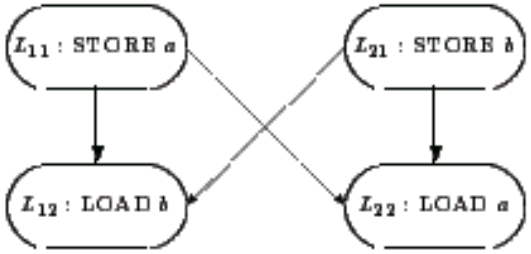
(a). $C(PRG_1)$

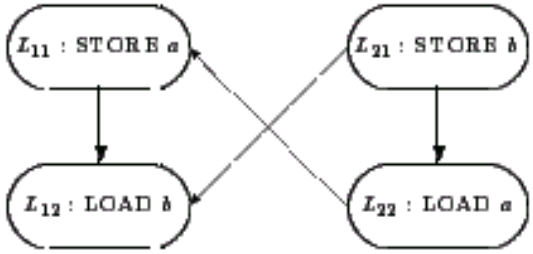


(b). $E_1(PRG_1)$ $(R_1 = 0, R_2 = 1)$



(c). $E_2(PRG_1)$ $(R_1 = R_2 = 0)$

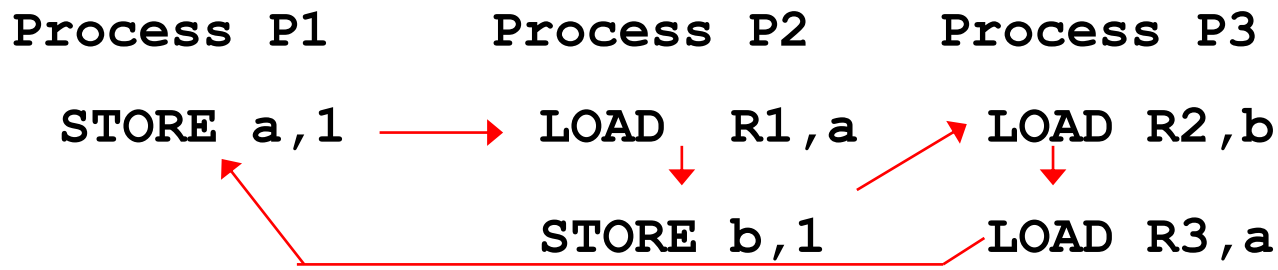




共享存储系统的访存事件次序

满足顺序一致的访存事件次序

- 顺序一致的一个充分条件（GPPO条件）
  - 在共享存储多处理机中，若任一处理机都严格按照访存指令在进程中出现的次序执行，且在当前访存指令彻底执行完之前不能开始执行下一条访存指令，则此共享存储系统是顺序一致的
  - 一个存数操作“彻底完成”指的是它所引起的值的变化已被所有处理机所接受
  - 一个取数操作彻底完成是指它取回的值已确定，且写此值的存数操作已“彻底完成”



分布式系统的访存模型

- 写可分割系统的访存事件模型
  - 系统中有N个处理机，每个处理机执行一个进程。
  - 任一访存操作u被分割成N个子操作$u^1, u^2, \ldots, u^n$，其中u^i表示u相对于Pi已执行完(performed with respect to Pi)。
- GPPO条件的描述
  - 任一处理机都按访存指令在进程中出现的次序执行，且在当前访存指令彻底执行完之前不能开始执行下一条访存指令

$$(u,v) \in PO \Rightarrow u^i < v^j$$

$$(w,r) \in E \cap (r,v) \in PO \Rightarrow w^i < v^j$$

$$(w1,w2) \in E \Rightarrow w1^i < w2^j$$

$$(r,w) \in E \Rightarrow r^c < w^c$$

$$(w,r) \in E \Rightarrow w^c < r^c$$

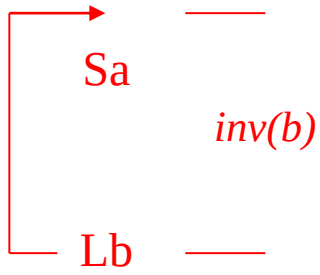
其中：c为执行操作的处理机号，i,j=1,2,…,n

- 在写可分割系统中，若执行E(PRG)满足GPPO条件，则E(PRG)正确

顺序一致系统中的乱序执行

- 在顺序一致系统中的乱序执行
  - 如果访存操作u和v是同一进程Pi的两个访存操作且u在v之前，则在如下条件下v可先于u执行而不影响程序正确性：在v发出之后u“彻底完成”之前的这段时间内，没有其他对v所访问单元的访问相对于Pi完成（如v所访问单元在Pi中的备份不被更新）
  - 例：

Process 1	Process2
STORE a,1	STORE b,1
LOAD R1,b	LOAD R2,a



- 乱序执行的实现方法：猜测执行
  - 先往下执行，收到相应的invalidate信号再反悔，区分处理器是不是顺序一致的主要标志

存储一致性模型

常见的存储一致性模型

- 常见的存储一致性模型
  - 顺序一致性模型（Sequential Consistency）
  - 处理器一致性模型（Processor Consistency）
  - 弱一致性模型（Weak Ordering）
  - 释放一致性模型（Release Consistency）
  - 域一致性模型（Scope Consistency）
  - 单项一致性模型（Entry Consistency）
- 从某种意义上说，存储一致性模型对共享存储系统中多处理机的访存次序作了限制，从而影响了性能

存储一致性模型（Memory Consistency）

- 历史的观点：Hardware-Centric
  - 对多个处理机访问共享存储器的次序的一些限制
  - 弱一致性模型：放松限制来提高性能
  - 程序员必须考虑访存次序
  - 系统设计者没有优化余地
- 正确的观点：Programmer-Centric
  - 结构设计者与应用程序员之间的一种约定
  - 给出正确程序的标准
  - 程序员不用考虑访存次序
  - 系统设计者有更多的提高性能的空间

顺序一致性模型

- 正确性标准：符合程序员的直觉
- 正确性规范：顺序一致性
  - Sequential consistency defines a correct execution as the one “whose result is the same as if the operations of each individual processor appear in this sequence in the order specified by the program”.
  - 如果在多处理机环境下的一个并行执行的结果等于同一程序在单处理机多进程环境下的一个执行的结果，则此并行程序执行正确

处理机一致性模型

- 由 Goodman 提出的处理机一致性（Processor Consistency）比顺序一致性弱，处理机一致性对访存事件发生次序施加的限制是：
  - 在任一取数操作 LOAD允许被执行之前，所有在同一处理机中先于这一LOAD 的取数操作都已完成
  - 在任一存数操作 STORE允许被执行之前，所有在同一处理机中先于这一STORE 的访存操作（包括LOAD 和STORE）都已完成
- 上述条件允许STORE之后的LOAD 越过STORE而执行，放松了顺序一致性模型对访存次序的限制
- 实际上是把Write Buffer变得让用户可见
  - 如：Store提交后在Write Buffer，还没有写Cache/内存，后面的Load已经从Write Buffer取回数据，此时收到对Load访问Cache行的一个无效请求

同步操作在共享存储系统中的作用

- 积分求π

```
jia_init(argc,argv);
pi=jia_alloc(8);
h = 1.0/N; pi = 0;
for(i=jiapid+1;i<=N;i+=jiahosts)
    { mypi = ... }
jia_lock(1);
    pi += mypi;
jia_unlock(1);
printf pi;
jia_exit();
```

- 矩阵乘法

```
double (*a)[N],(*b)[N],(*c)[N];
a=jia_alloc(N*N*8);
b=jia_alloc(...); c=jia_alloc(...);
if (jiapid==0) for (i...) for (j…){
  a[i][j]=1;b[i][j]=1;
}
jia_barrier();
begin=N*jiapid/jiahosts;
end=N*(jiapid+1)/jiahosts;
for (i=begin; i<end; i++)
    for (j=0; j<N; j++)
          for (k=0; k<N; k++)
            c[i][j]+=a[i][k]*b[k][j];
jia_barrier();
if (jiapid==0) printf C;
jia_exit();
```

弱存储一致性模型（Weak Ordering）

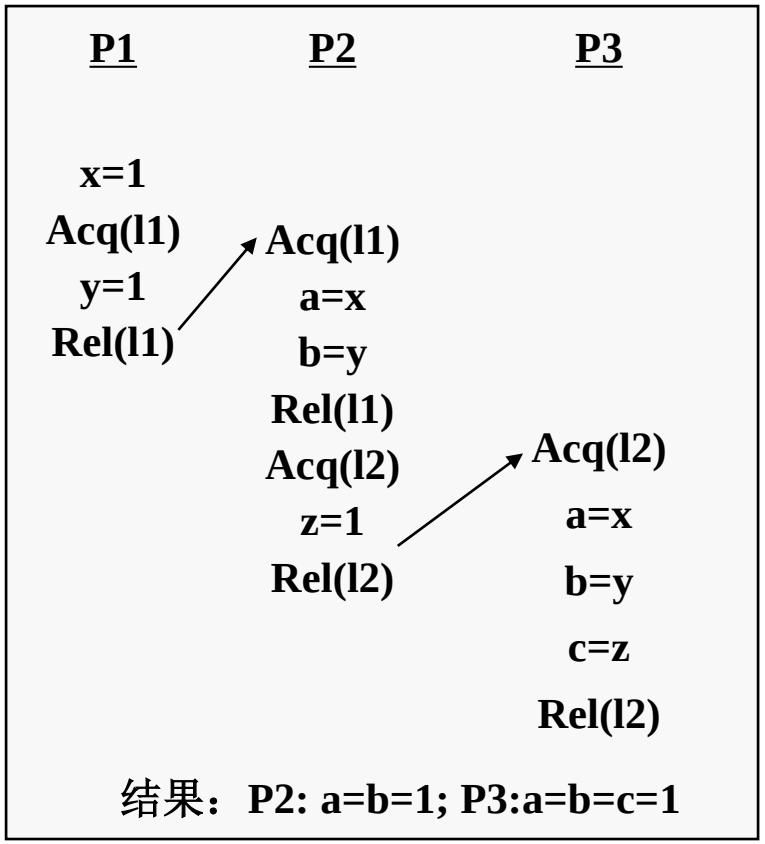
- 把同步操作和普通访存操作区分开来，只在同步点维护一致性
  - 同步操作实现顺序一致性
  - 冲突访问必须用同步操作保护
- 访存次序：
  - 同步操作的执行满足顺序一致性条件
  - 在任一普通访存操作允许被执行之前，所有在同一处理机中先于这一访存操作的同步操作都已完成
  - 在任一同步操作允许被执行之前，所有在同一处理机中先于这一同步操作的普通访存操作都已完成

释放一致性模型（Release Consistency）

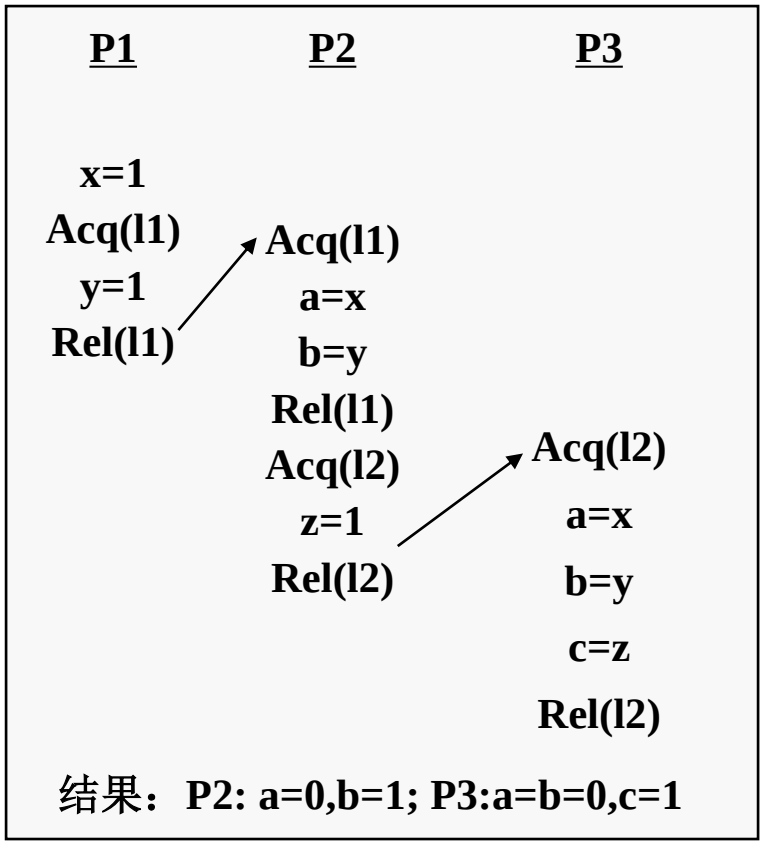
- 把同步操作进一步分成获取操作ACQUIRE和释放操作RELEASE
- 冲突访问必须用REL->ACQ对隔开
- 访存次序
  - 同步操作的执行满足顺序一致性条件
  - 在任一普通访存操作允许被执行之前，所有在同一处理机中先于这一访存操作的ACQUIRE操作都已完成
  - 在任一RELEASE操作允许被执行之前，所有在同一处理机中先于这一RELEASE的普通访存操作都已完成

域一致性模型（Scope Consistency）

- 冲突访问必须用同一把锁保护



Release Consistency



Scope Consistency

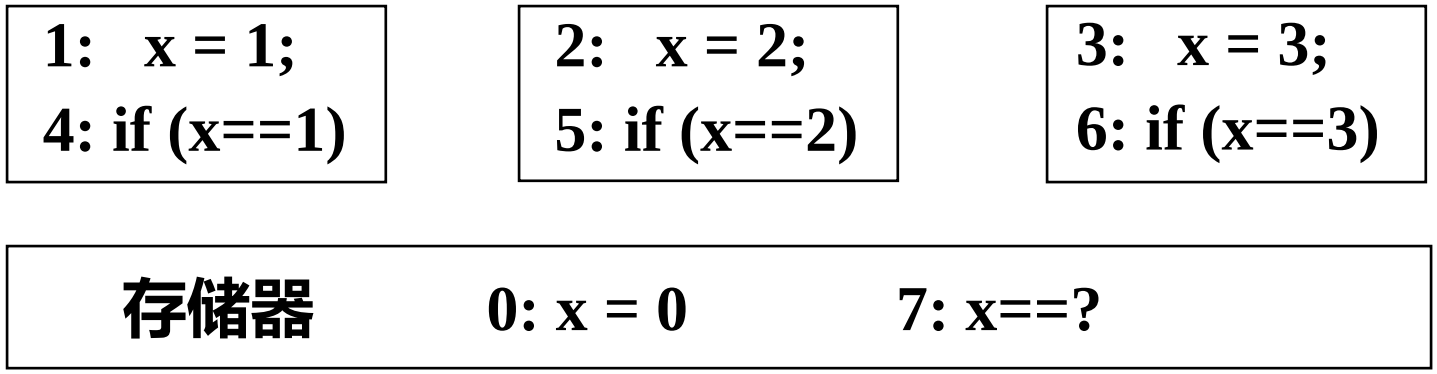
单项一致性模型（Entry Consistency）

- 必须为每一个共享变量指定相应的锁
- 加重了程序员的负担
- 有利于性能

CACHE一致性协议

共享存储多处理机中的Cache一致性问题

- Cache在共享存储系统中的作用
  - 弥补CPU与主存间的速度差距
  - 减少访存冲突以及对互连网络带宽的需求
- Cache一致性问题
  - 如何保持数据在Cache及主存中的多个备份的一致性



Cache一致性协议

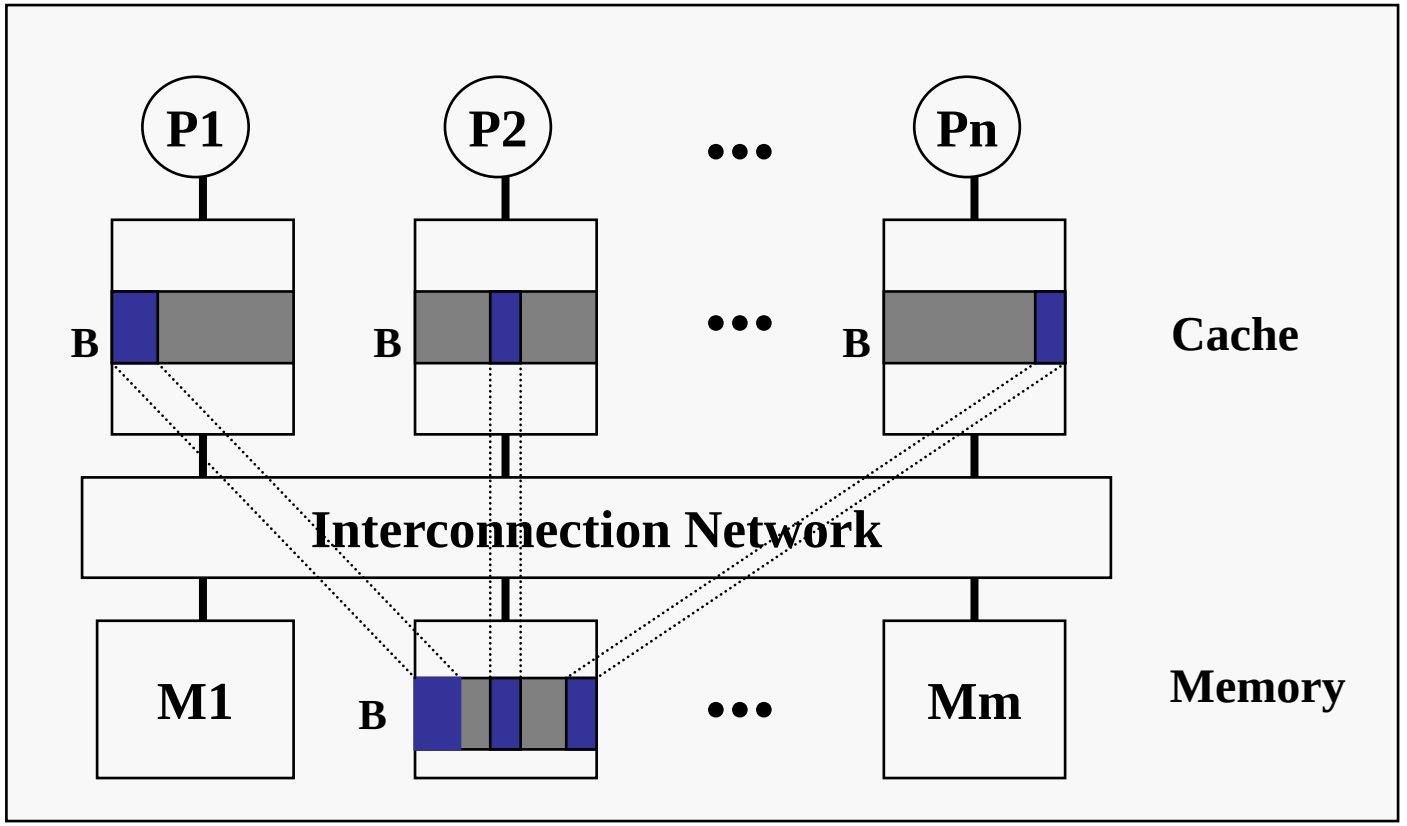
- Cache一致性协议（Cache Coherence Protocol）：一种把新写的值传播到其他处理机的机制
  - 如何传播新值：Write Invalidate vs. Write Update
  - 谁可以产生新值：Single Writer vs. Multiple Writer
  - 何时开始传播新值：Early vs. Delayed Propagation
  - 向何处传播新值：Snoopy vs. Directory Protocol
- Cache一致性协议决定系统为维护一致性所做的具体动作，因而直接影响系统性能

Write-Invalidate和Write Update

- Write-Invalidate
  - 当一个处理机更新某共享单位（如存储行或存储页）时（之前或之后），通过某种机制使该共享单位的其它备份无效，当其它处理机访问该共享单位时，访问失效，再取得有效备份
- Write-Update
  - 当一个当一个处理机更新某共享单位时，把更新的内容传播给所有拥有该共享单位备份的处理机
- 比较
  - Write Update: 重复更新已不再使用的行
  - Write Invalidate: 假共享导致乒乓问题

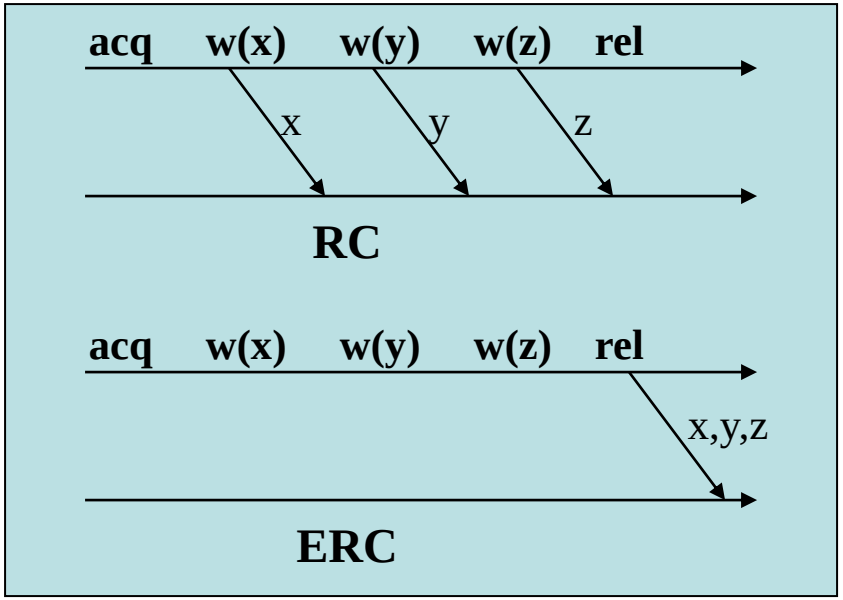
单写和多写

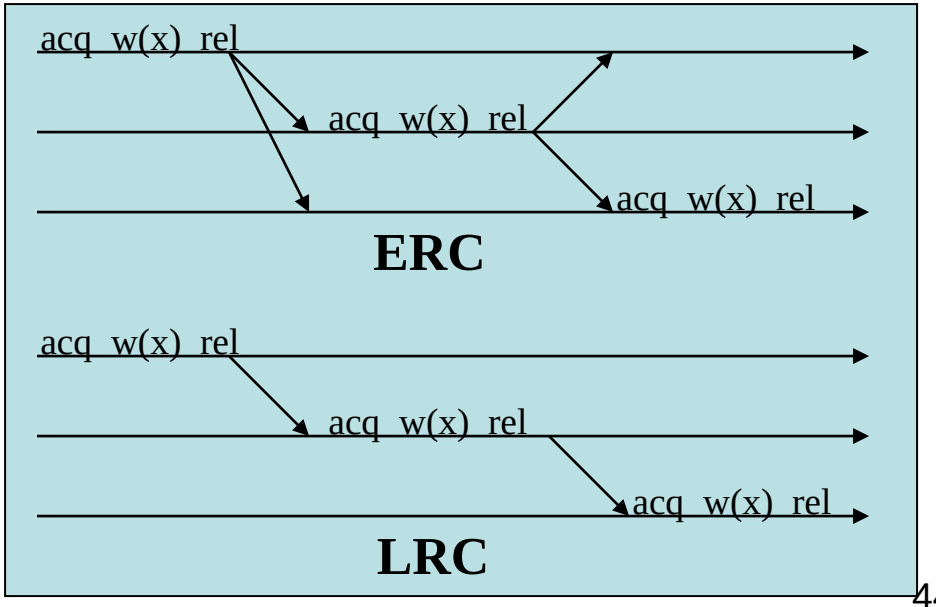
- 单写：任一时刻只有一个处理机能写某共享单位
- 多写：多个处理机同时写某共享单位的不同部分



急切更新与懒惰更新

- 普通RC：边写边传输写操作
- Eager RC：把写操作延迟到Release时一起发出
- Lazy RC：进一步延迟到下一个Acquire时再传输
- ERC和LRC是RC的不同实现，不是不同的一致性模型





侦听协议

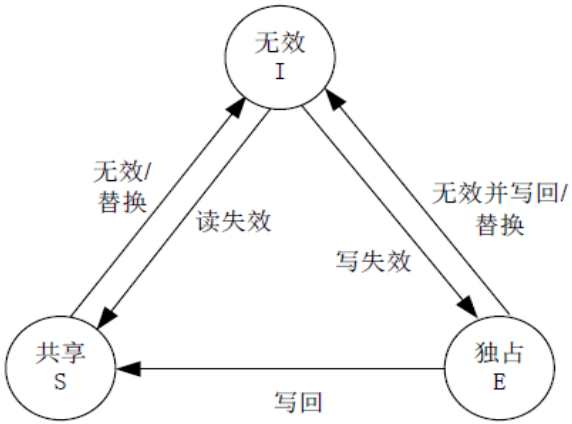
- 通过广播维护一致性
  - 写数的处理机把新写的值或所需的存储行地址广播出去
  - 其他处理机侦听广播，当广播中的内容与自己有关时，接受新值或提供数据
  - 存储器和每个处理机的Cache只维护状态信息
- 适合于总线结构的SMP系统中
  - 总线是一种廉价而有效的广播工具
- 可伸缩性有限
  - 总线是一种独占性资源
  - 总线延迟随处理机数的增加而增加：仲裁、总线长度、总线阻抗

基于目录的协议

- 每个存储行对应一个目录项
  - 记录拥有该行的一个副本的那些处理机
  - 当某个处理机写该行时,根据目录项的内容传播数据
  - 在COMA结构中，记录该行的Owner
- 由于避免了广播，目录协议有一定的伸缩性
- 目录需要大量存储空间，需要动态维护
  - 位向量目录：O(MN)，存储开销大
  - 有限指针目录：O(M logN)，指针溢出
  - 链目录：O(M logN)，串行更新

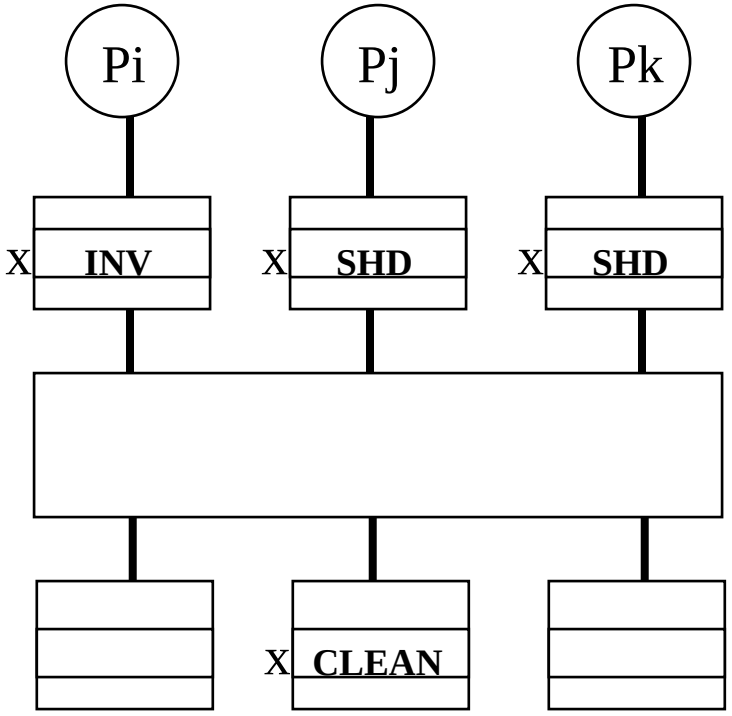
ESI协议的状态

- ESI 是指Cache 行的三种一致性状态：
  - E（Exclusive，独占）：表明对应Cache 行被当前处理器核独占，当前处理器核可以随意读写，其他处理器核如果想读写这个cache 行需要请求占有这个cache 块的处理器核释放该Cache 行
  - S（Shared，共享）：表明当前Cache 行可能被多个处理器核共享，只能读取，不能写入
  - I （Invalid，无效）：表明当前Cache 块是无效的

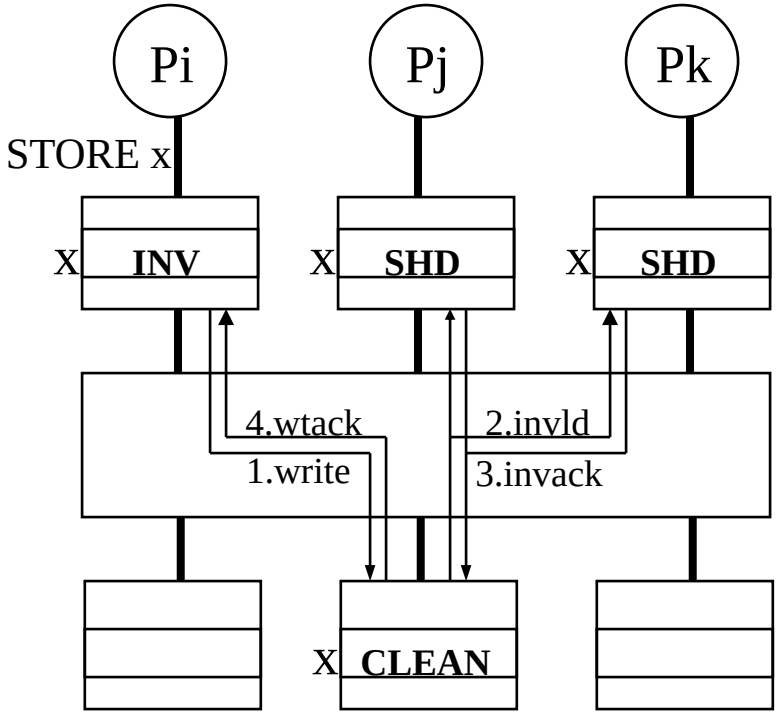


协议举例

- 基于目录，单写，写使无效

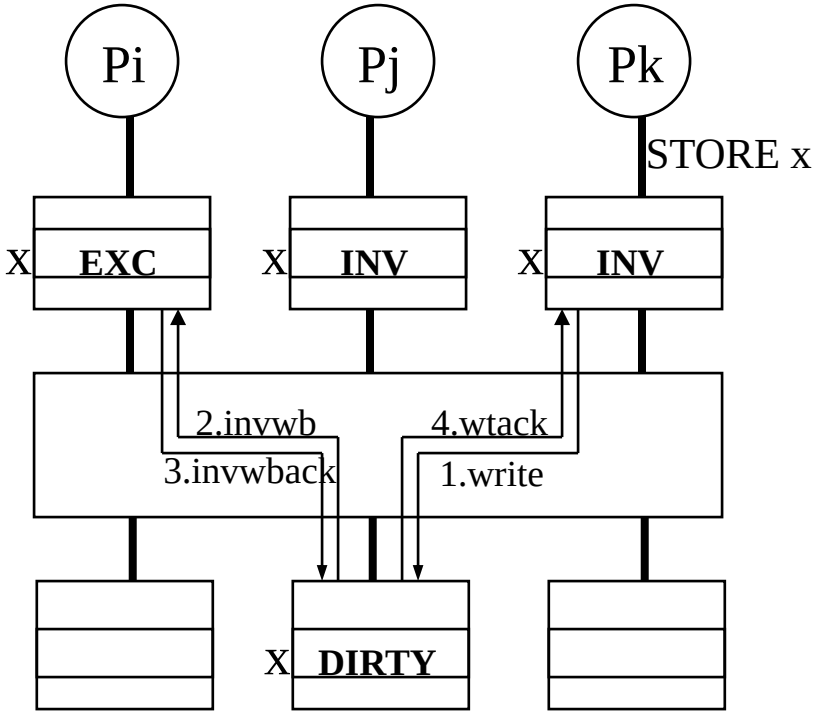


(a)初始状态

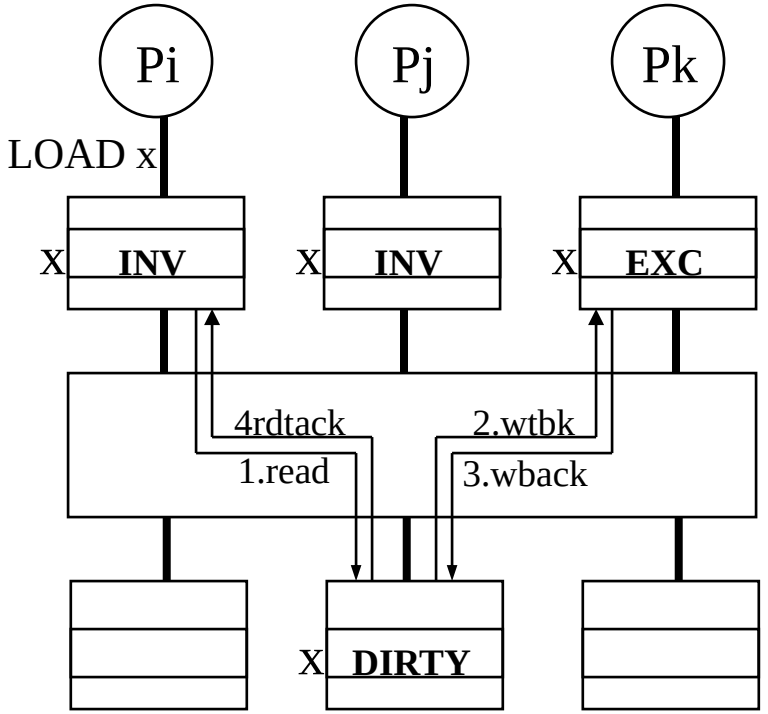


(b)Pi发出存数操作

协议举例（cont.）



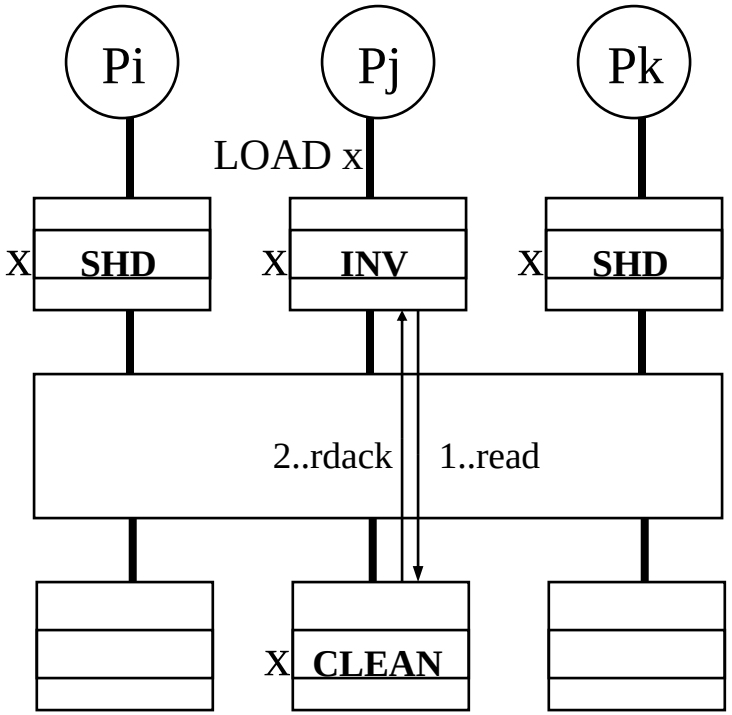
(c)Pk发出存数操作



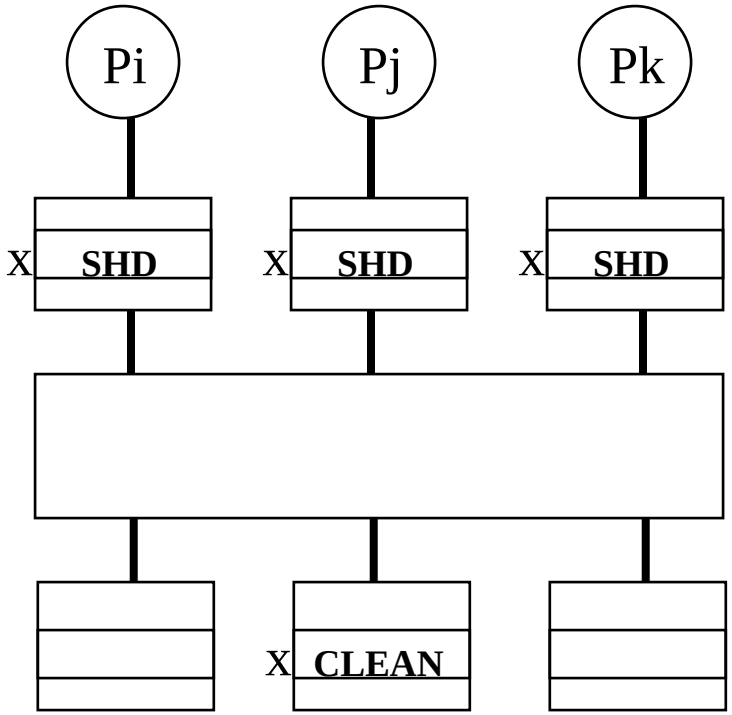
(d)Pi发出存数操作

- **若支持SC，Pi需等到Pk的“STORE x”执行完**
- **若支持RC，无限制**

协议举例（cont.）



(e)Pj发出取数操作



(e)最终状态

存储一致性模型与Cache一致性协议

- 存储一致性模型对Cache一致性协议的制约作用
  - Cache一致性协议都是针对某种存储一致性模型而设计的
  - 存储一致性模型为Cache一致性协议规定了“一致性”的目标，即，什么是“一致性”
  - 如，顺序一致性模型要求对某处理机所写的值立即进行传播，在确保该值已经被所有处理机接受后才能继续其它指令的执行
  - 如，释放一致性模型允许将某处理机所写的值延迟到释放锁时进行传播

访存事件次序在微结构中的实现

- 指令在保留站中等待发射时检查并等待
  - 等到前面所有指令都提交
  - Store操作写到Cache、访存失效队列空
  - 同步操作如sync、LL、SC一般在发射时控制
- Load操作能不能越过未完成（地址未确定或尚在Store Buffer中）的Store操作执行
  - 如果Load数据返回到寄存器并被使用了，取消起来很麻烦
  - 外部来的invalidate操作要查看Store Buffer也很麻烦
- 为了性能和实现的简洁常使用弱一致性模型

不忘初心：提高性能

- 串行程序提高性能
  - 保持程序相关性基础上打乱执行次序
  - 保留站、寄存器重命名、ROB
  - 指令级并行不改变编程模型
- 并行程序提高性能
  - 除了线程内部相关性，还要考虑线程间相关性
  - 弱一致性模型，分布式共享存储，目录一致性协议
  - 弱一致性模型利用共享并行程序的同步实现一致性，只在同步点维持顺序执行，普通操作可以像单核一样乱序执行，对部分程序编程有一定影响

分析多核CPU的要素

- 处理器核
  - 异构、同构；通用、专用；重核、轻核；多核、众核
- 互连结构
  - 可伸缩程度，是否支持片间互连
- 访存结构
  - 所有核同一地址空间、不同核不同地址空间
  - Cache一致性协议：是否可伸缩
- 峰值性能和访存带宽及IO带宽
  - 避免茶壶里面倒饺子
  - 通用多核处理器访存带宽和峰值性能差距不能太大

参考文献

[1] 胡伟武, 《共享存储系统结构》, 高等教育出版社, 2001

[2] Weiwu Hu, “Correct Event Ordering in Shared Memory Syetems”, Ph.D. Thesis, Institute of Computing Technology, CAS, March, 1996.

[3] Weiwu Hu and Peisu Xia, “Out-of-Order Execution in Sequentially Consistent Shared Memory Systems: Theory and Experiments, *Journal of Computer Science and Technology*, Vol. 13, No. 2, pp. 125-140, Mar. 1998.

[4] Weiwu Hu, Weisong Shi, and Zhimin Tang, “A Framework of Memory Consistency Models”, *Journal of Computer Science and Technology*, Vol. 13, No. 2, pp. 110-124, Mar. 1998.

作 业