高等计算机系统结构

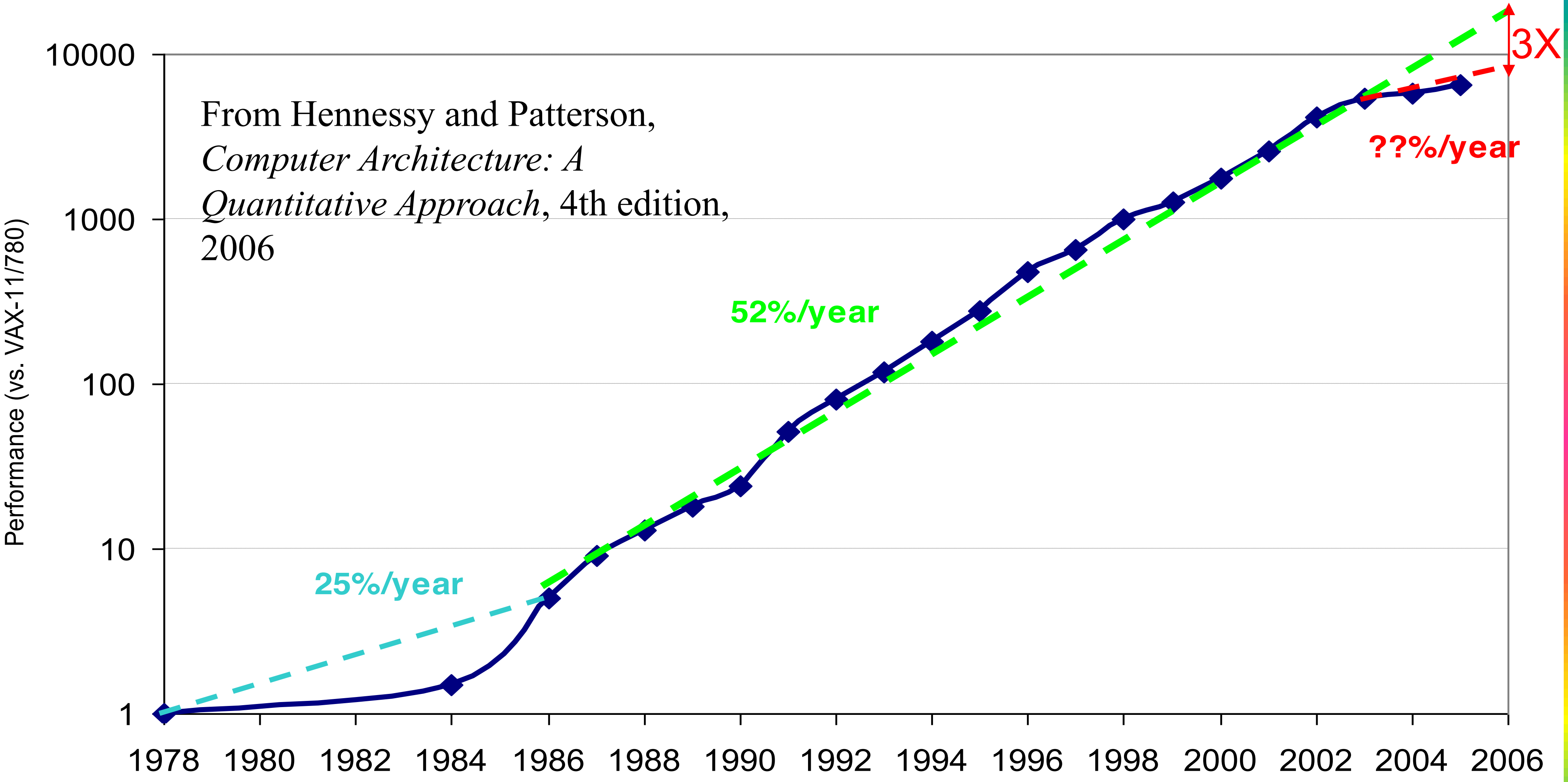
多处理器系统

（第九讲）

程 旭

2014年5月12日

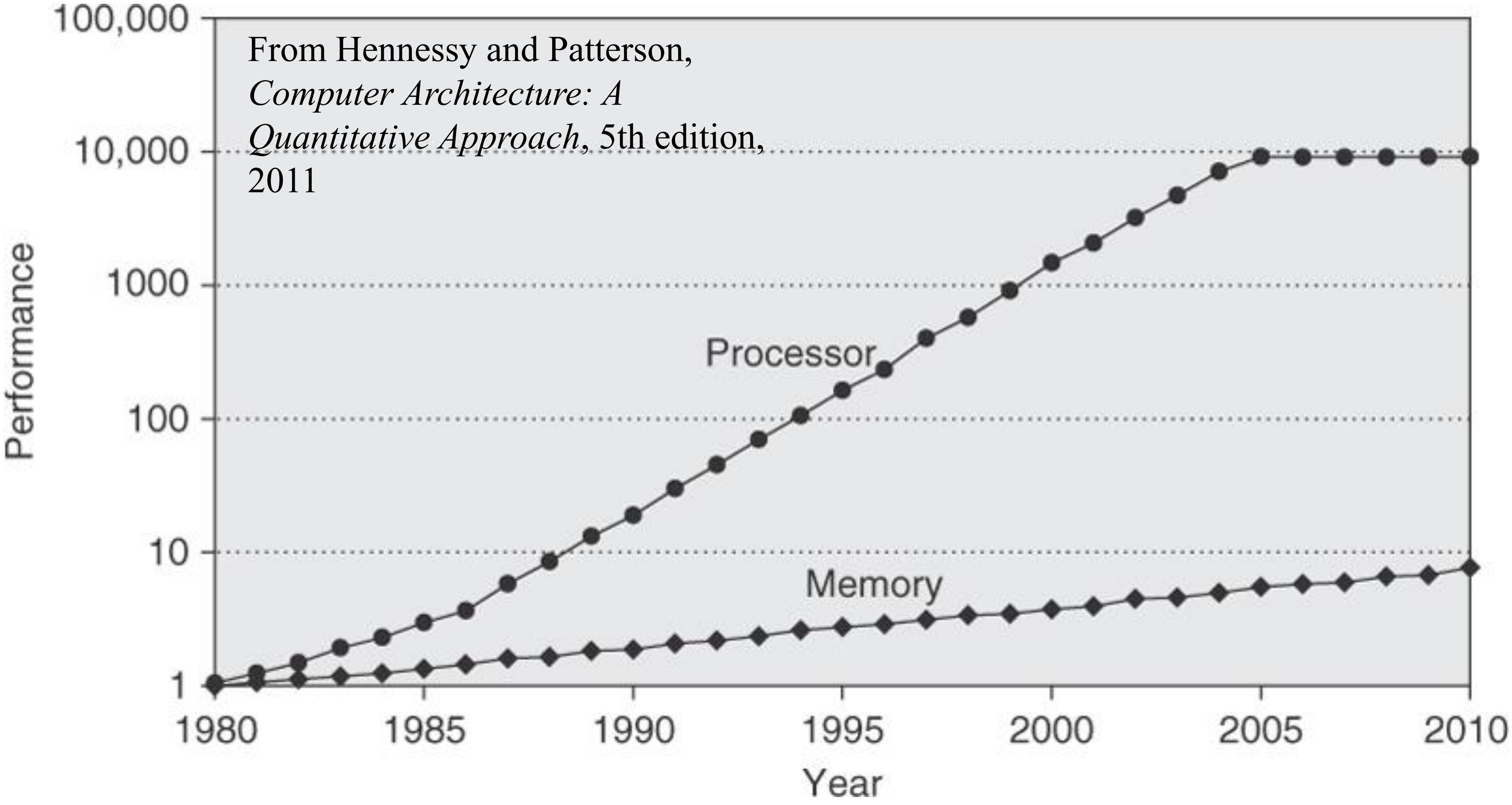
Uniprocessor Performance (SPECint)



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, 2006

- **VAX : 25%/year 1978 to 1986**
- **RISC + x86: 52%/year 1986 to 2002**
- **RISC + x86: ??%/year 2002 to present**

Uniprocessor Performance



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th edition, 2011

Parallel Processing:
Déjà vu all over again?

- "… today's processors … are nearing an impasse as technologies approach the speed of light.."
  - David Mitchell, The Transputer: The Time Is Now (1989)
- Transputer had bad timing (Uniprocessor performance↑)
  ⇒ Procrastination rewarded: 2X seq. perf. / 1.5 years
- "We are dedicating all of our future product development to multicore designs. … This is a sea change in computing"
  - Paul Otellini, President, Intel (2005)
- All microprocessor companies switch to MP (2+ CPUs/2 yrs)
  ⇒ Procrastination penalized: 2X sequential perf. / 5 yrs
- Even handheld systems moved to multicore
  - Nintendo 3DS, iPhone4S, iPad 3 have two cores each (plus additional specialized cores)
  - Playstation Portable Vita has four cores

Other Factors ⇒ Multiprocessors

- Growth in data-intensive applications
  - Data bases, file servers, …
- Growing interest in servers, server perf.
  - cloud computing and software-as-a-service
- Increasing desktop perf. less important
  - Outside of graphics
- Improved understanding in how to use multiprocessors effectively
  - Especially server where significant natural TLP
- Advantage of leveraging design investment by replication
  - Rather than unique design

并行计算机

- 定义： “A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”

  Almasi and Gottlieb, *Highly Parallel Computing* ,1989

- 并行计算机的一些问题:
  - 资源分配:
    - 处理单元有多少？
    - 处理单元的性能如何？
    - 存储多大?
  - 数据访问、通信和同步
    - 处理单元之间如何协作和通信？
    - 互联是什么类型?
    - 数据如何在处理器之间传输？
    - 编程人员使用什么样的原语？
  - 性能和可扩展性
    - 上述因素如何影响性能?
    - 如何支持可扩展性?

并行处理器的 “信仰”

- 六十年代以来计算机设计人员的梦想：增加处理器数量以提升性能 与 设计更快的处理器
- 由于“单处理器不能继续发展”，因而导致创造出许多针对具体编程模型的机器组成
  - 例如，由于受制于光速限制，单处理器的速度将不再提升： **1972, … , 1989**
  - 近乎宗教的狂热：必须确信无疑！
  - 九十年代，一些著名公司，如 **Thinking Machines**、**Kendall Square, …**等退出商业领域，这种狂热有所降温
- 论据变为：可扩展性能机遇的“拉动”，而非“单处理器性能稳定”的“推动”

什么级别的并行性?

- 位级并行性: 1970 到 1985左右
  - 4位、8位、16位、 32位处理器
- 指令级并行 (ILP): 1985 到 今天
  - 流水技术
  - 超标量
  - 超长指令字
  - 乱序执行
  - 指令级并行性的限制?
- 进程级 或 线程级并行性

What is *Parallel* Architecture?

- A parallel computer is a collection of processing elements that cooperate to solve large problems
  - Most important new element: It is all about communication!
- What does the programmer (or OS or Compiler writer) think about?
  - Models of computation:
    - PRAM? BSP? Sequential Consistency?
  - Resource Allocation:
    - how powerful are the elements?
    - how much memory?
- What mechanisms must be in hardware vs software
  - What does a single processor look like?
    - High performance general purpose processor
    - SIMD processor/Vector Processor
  - Data access, Communication and Synchronization
    - how do the elements cooperate and communicate?
    - how are data transmitted between processors?
    - what are the abstractions and primitives for cooperation?

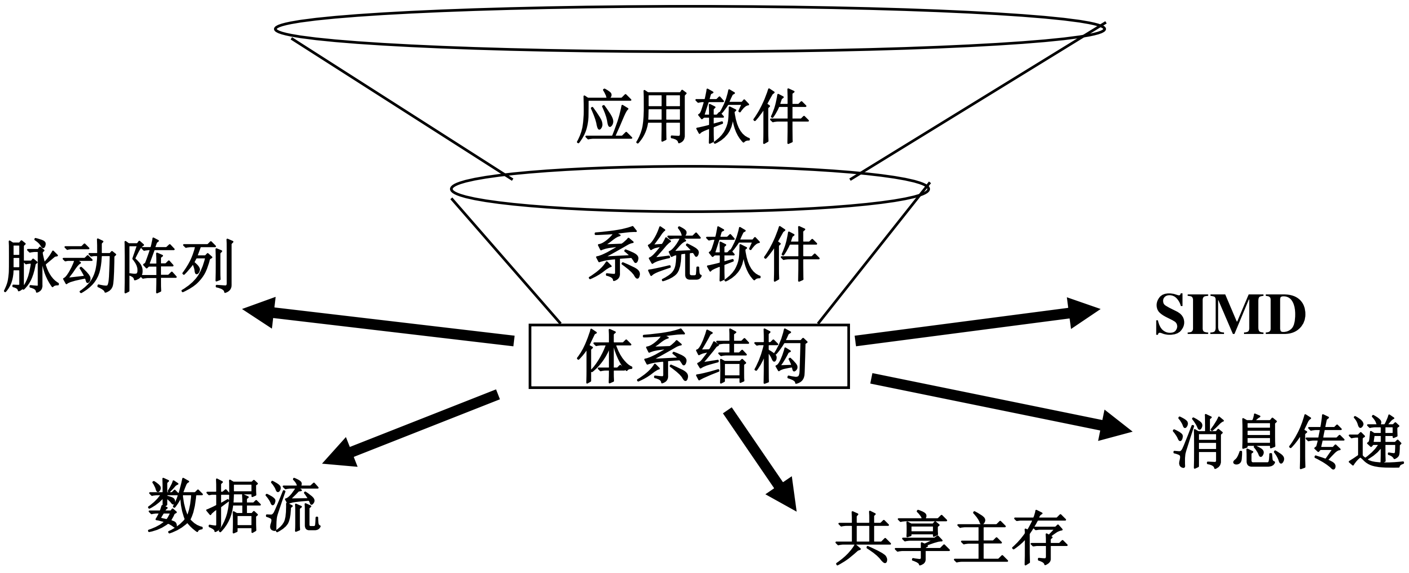
并行体系结构

- 并行体系结构用通信体系结构（communication architecture ）对传统的体系结构进行扩展
  - 抽象 (硬件/软件接口)
  - 组成结构支持有效实现上述抽象

并行体系结构历史

历史上，并行体系结构与编程模型紧密结合

- 出现大量不同的结构，不可预测到底会如何发展



- 发展方向的不确定性严重影响了并行软件的开发!

Parallel Programming Models

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - What orderings exist between operations?
  - How do different threads of control synchronize?
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations can be used to coordinate parallelism
  - What are the atomic (indivisible) operations?
- Cost
  - How do we account for the cost of each of the above?

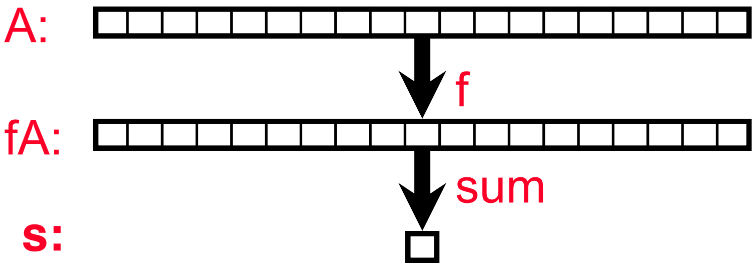
Simple Programming Example

- Consider applying a function f to the elements of an array A and then computing its sum:

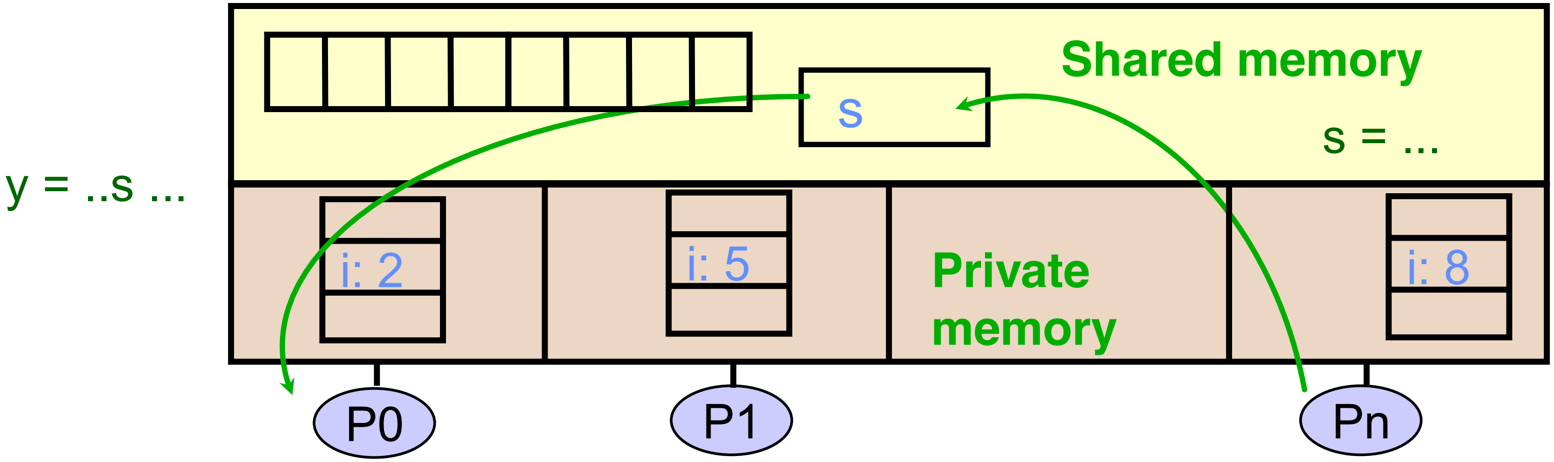
$$\sum_{i=0}^{n-1} f(A[i])$$

- Questions:
  - Where does A live? All in single memory? Partitioned?
  - What work will be done by each processors?
  - They need to coordinate to get a single result, how?

A = array of all data
fA = f(A)
s = sum(fA)



Programming Model 1: Shared Memory

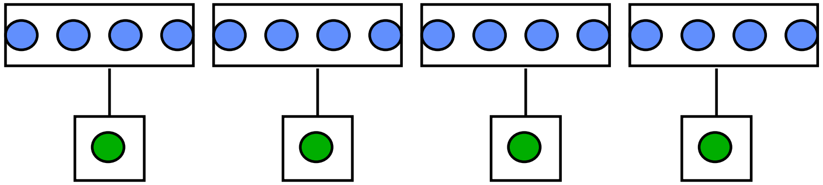


- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate **implicitly** by writing and reading shared variables.
  - Threads coordinate by **synchronizing** on shared variables

Simple Programming Example: SM

$$\sum_{i=0}^{n-1} f(A[i])$$

- **Shared memory strategy:**
  - small number p << n=size(A) processors
  - attached to single memory
- **Parallel Decomposition:**
  - Each evaluation and each partial sum is a task.
- **Assign n/p numbers to each of p procs**
  - Each computes independent “private” results and partial sum.
  - Collect the p partial sums and compute a global sum.



Two Classes of Data:

- **Logically Shared**
  - The original n numbers, the global sum.
- **Logically Private**
  - The individual function evaluations.
  - What about the individual partial sums?

Shared Memory “Code” for sum

```
static int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + f(A[i])
```

```
Thread 2

  for i = n/2, n-1
      s = s + f(A[i])
```

Problem is a race condition on variable s in the program

A race condition or data race occurs when:

- two processors (or two threads) access the same variable, and at least one does a write.
- The accesses are concurrent (not synchronized) so they could happen simultaneously

A Closer Look

A	3	5

f = square

static int s = 0;

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + reg0	9	reg1 = reg1 + reg0	25
s = reg1	9	s = reg1	25
…		…	

Assume A = [3,5], f is the square function, and s=0 initially

For this program to work, s should be 34 at the end

- but it may be 34,9, or 25

The *atomic* operations are reads and writes

- Never see ½ of one number, but += operation is not atomic
- All computations happen in (private) registers

Improved Code for Sum

```
static int s = 0;
static lock lk;
```

```
Thread 1

    local_s1= 0
    for i = 0, n/2-1
        local_s1 = local_s1 + f(A[i])
    lock(lk);
    s = s + local_s1
    unlock(lk);
```

```
Thread 2

    local_s2 = 0
    for i = n/2, n-1
        local_s2= local_s2 + f(A[i])
    lock(lk);
    s = s +local_s2
    unlock(lk);
```

Since addition is associative, it’s OK to rearrange order

Most computation is on private variables

- Sharing frequency is also reduced, which might improve speed
- But there is still a race condition on the update of shared s

- The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

What about Synchronization?

- All shared-memory programs need synchronization
- Barrier – global (/coordinated) synchronization
  - simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier;
read_neighboring_values();
barrier;
```

- Mutexes – mutual exclusion locks
  - threads are mostly independent and must access common data

```
lock *l = alloc_and_init();    /* shared */
lock(l);
 access data
unlock(l);
```

- Need atomic operations bigger than loads/stores
  - Actually – Dijkstra’s algorithm（p-v） can get by with only loads/stores, but this is quite complex (and doesn’t work under all circumstances)
  - Example: atomic swap, test-and-set
- Another Option: Transactional memory 这个并不是课程重点
  - Hardware equivalent of optimistic concurrency
  - Some think that this is the answer to all parallel programming

并行构架

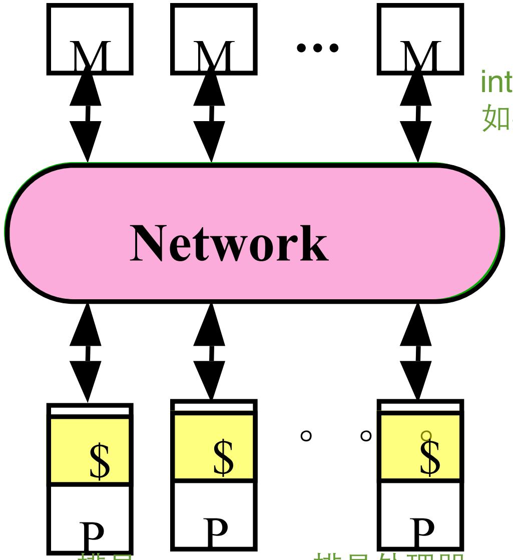
MIMD的机器可以分为两类
一类是集中式共享存储结构，常用于较小的系统，常用总线等方式将各个处理器连接起来，比较容易做到各处理器访存一致，因此常成为UMA。
一类是分布式共享存储结构，常用于处理器数量较多的系统，这时无法做到UMA，就称为NUMA，常常是每个处理器自带一个本地的memory。NUMA又有两种通信抽象，一种是所有处理器各自带的memory统一编制，全局可见，都通过store/load指令来存取，另一种是一个处理器智能看到它自己带的memory，想跟别的处理器交换数据必须用send/receive
第一类称为共享存储机器，第二类成为消息传递机器

- 分层：
  - 编程模型
    - 多道程序（Multiprogramming）：许多工作之间没有通信
    - 共享地址空间：通过存储器通信
    - 消息传递：发送 和 接收 信息（信报）
    - 数据并行： 多个代理同时对不同的数据集合进行操作，然后同时在全局交换信息 (共享或消息传递)
  - 通信抽象：

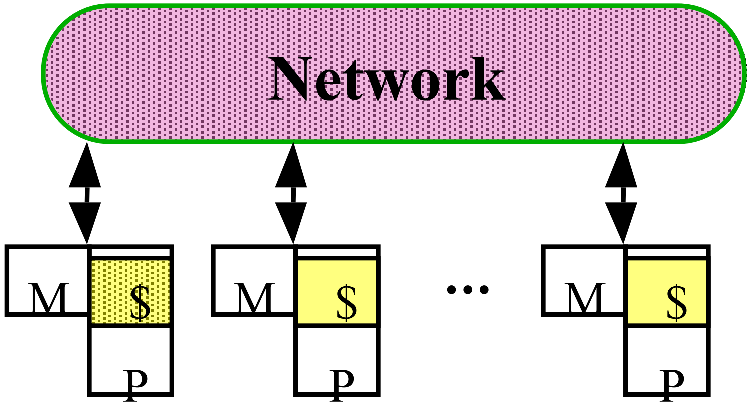
跟早期讲数组的SMD有异曲同工之妙，是从那里派生出来的。我们做科学计算大部分都是数组操作数组元素类型和操作相同，仅仅数值不同而已，我们可以把一个指令加到不同的数据上，这叫SMD就让一个处理器在自己的数据集合上做这种SMD的操作，过一会儿再统一交换一下信息。

- 共享地址空间：例如，load, store, atomic swap
- 消息传递：例如，send, receive的库调用
- 关于这一论题的争论 （易于编程、可扩展能力）

可扩展性



interconnnected network 互联网络，说不同机器之间都有互联，而不是只有一个bus大家一起联
如果用bus，那数据量一大就很有瓶颈



一排是memory，一排是处理器

“舞厅式”

自己找自己的memory

“闺房式”

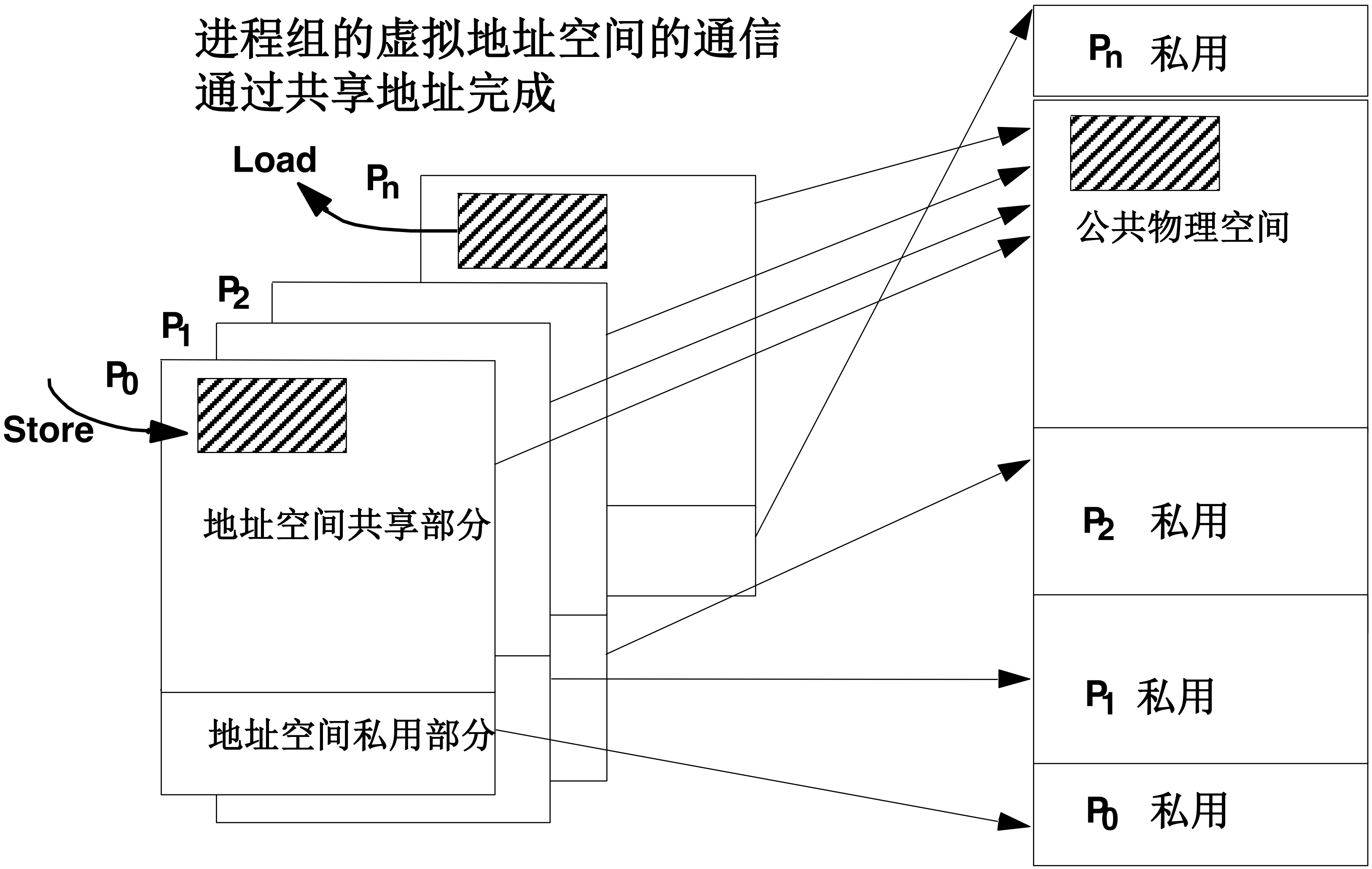
- 互联网络是问题所在: 成本 (交叉开关) 或带宽(总线)
- 舞厅式：带宽仍可扩展，但比交叉开关的成本更低
  - 到存储器的时延是统一的，但统一为最大时延
- 分布存储器 和 非统一存储器访问（non-uniform memory access：NUMA）
  - 在通用网络上，构造成简单的信报（消息）事务之外的共享地址空间 (例如，读请求（read-request）、读响应（read-response））
- 高速缓存共享（特别是非本地）的数据?

共享地址模型小结

- 每个 处理器 可以指定（name）该机器中所有的物理位置
- 每个 进程 可以指定（name）它与其他进程共享的所有数据
- 数据通过 load和store传输
- 数据大小：字节、字、 ... 或 cache块
- 使用虚拟存储技术来将虚拟地址映射到本地或远程的物理地址
- 存储层次模型要求：通信将数据移动到本地处理器的cache (就象load把数据从存储器移动到cache)
  - 通信时，时延、带宽、可扩展性?

共享地址空间模型

进程组的虚拟地址空间的通信
通过共享地址完成

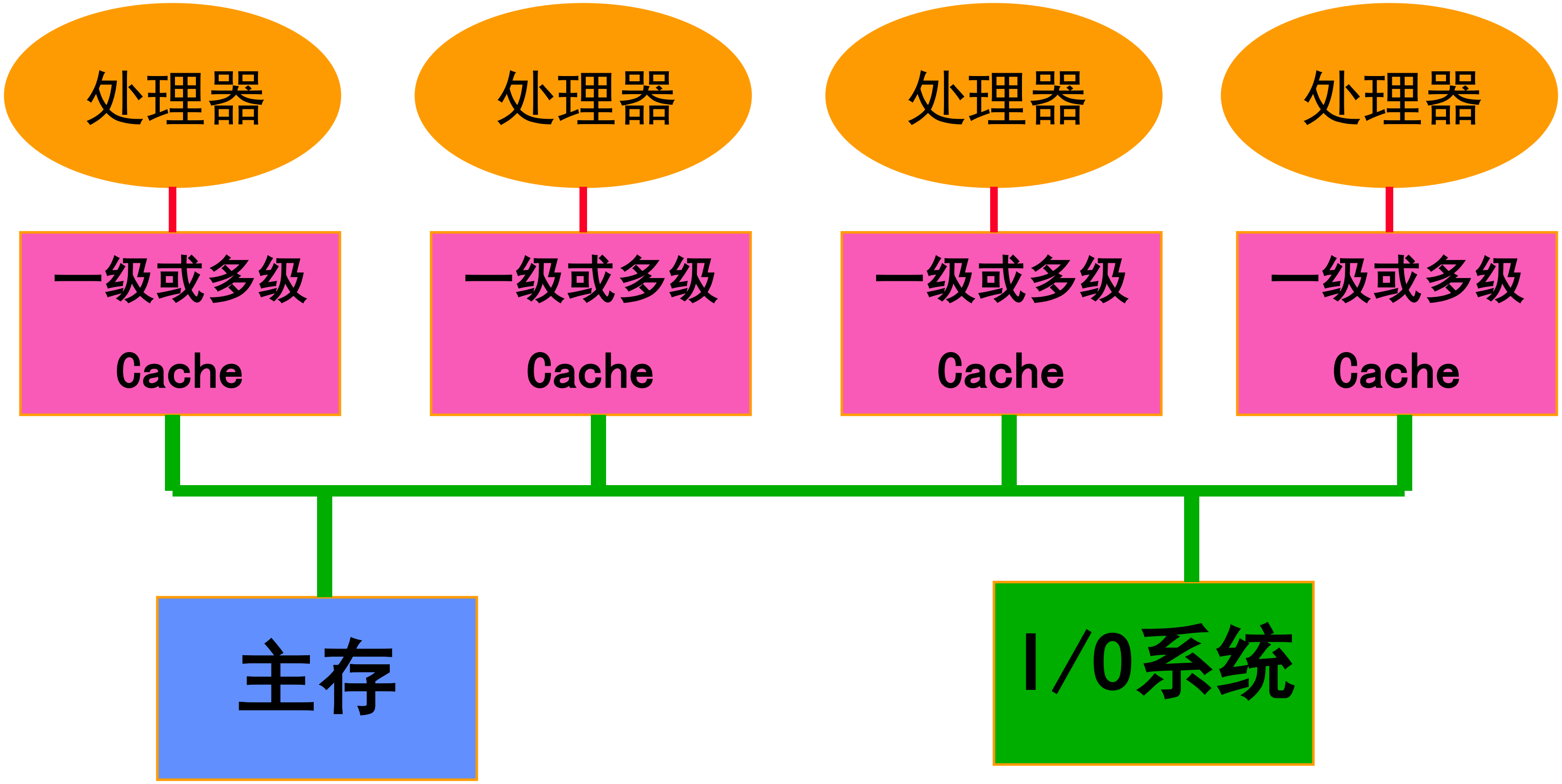


共享地址/存储多处理器模型

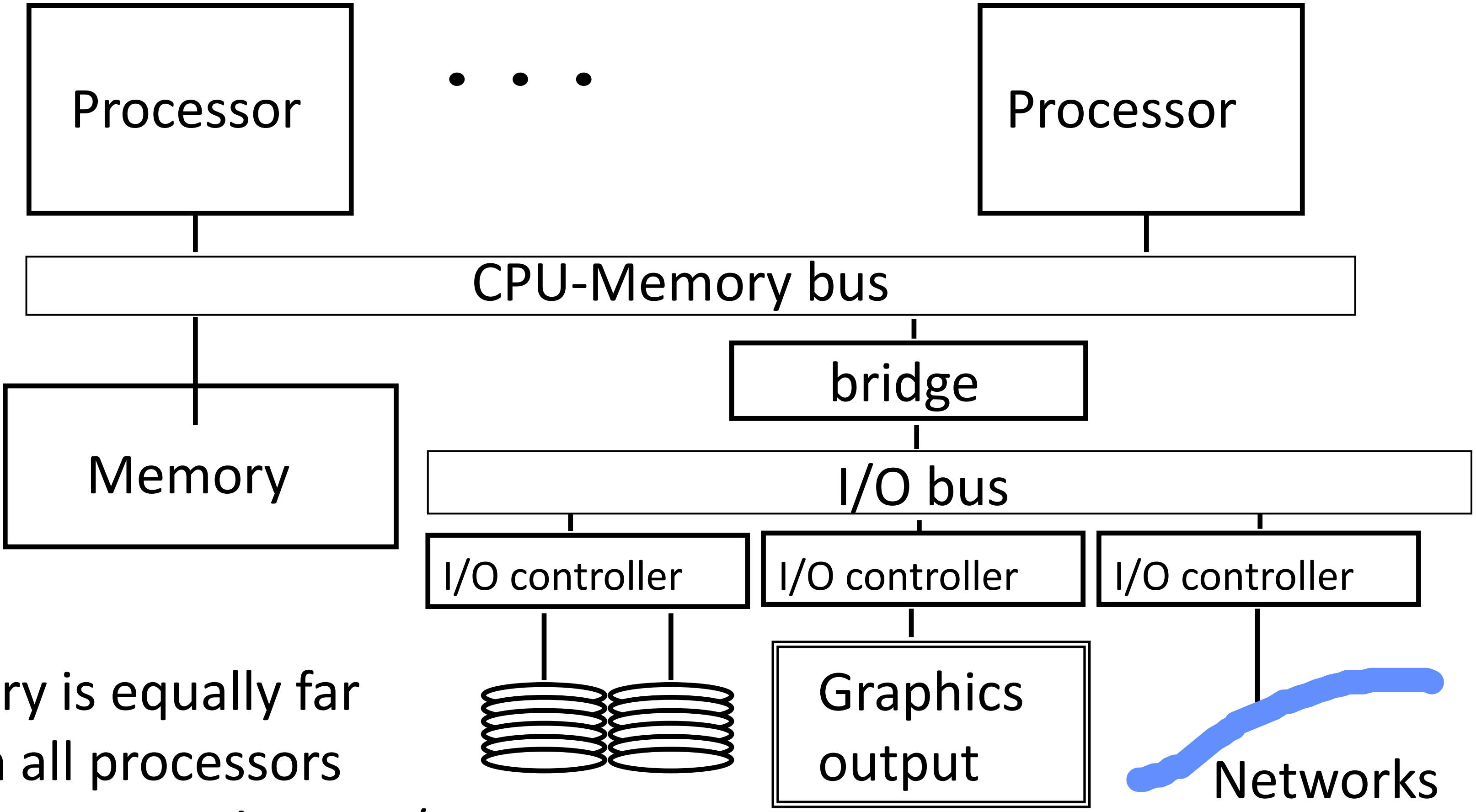
- 通过 Load 和 Store 通信
  - 最老的、也是使用最广的模型
- 基于时间共享：多处理器上的多进程 与 共享单处理器
- 进程：单一虚拟地址空间 和 单或多线程控制
  - 多进程可以重叠 (共享)，但是所有 线程 共享同一进程地址空间
- 一个线程对共享地址空间的写操作对其他线程的读操作是可见的
  - 通常模型：共享代码、私有栈、一些共享堆、和一些私用堆

示例：小规模多处理器设计

- 存储器：具有相同访问时间（uniform access time：UMA） 和 总线互联、I/O



Symmetric Multiprocessors



symmetric

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

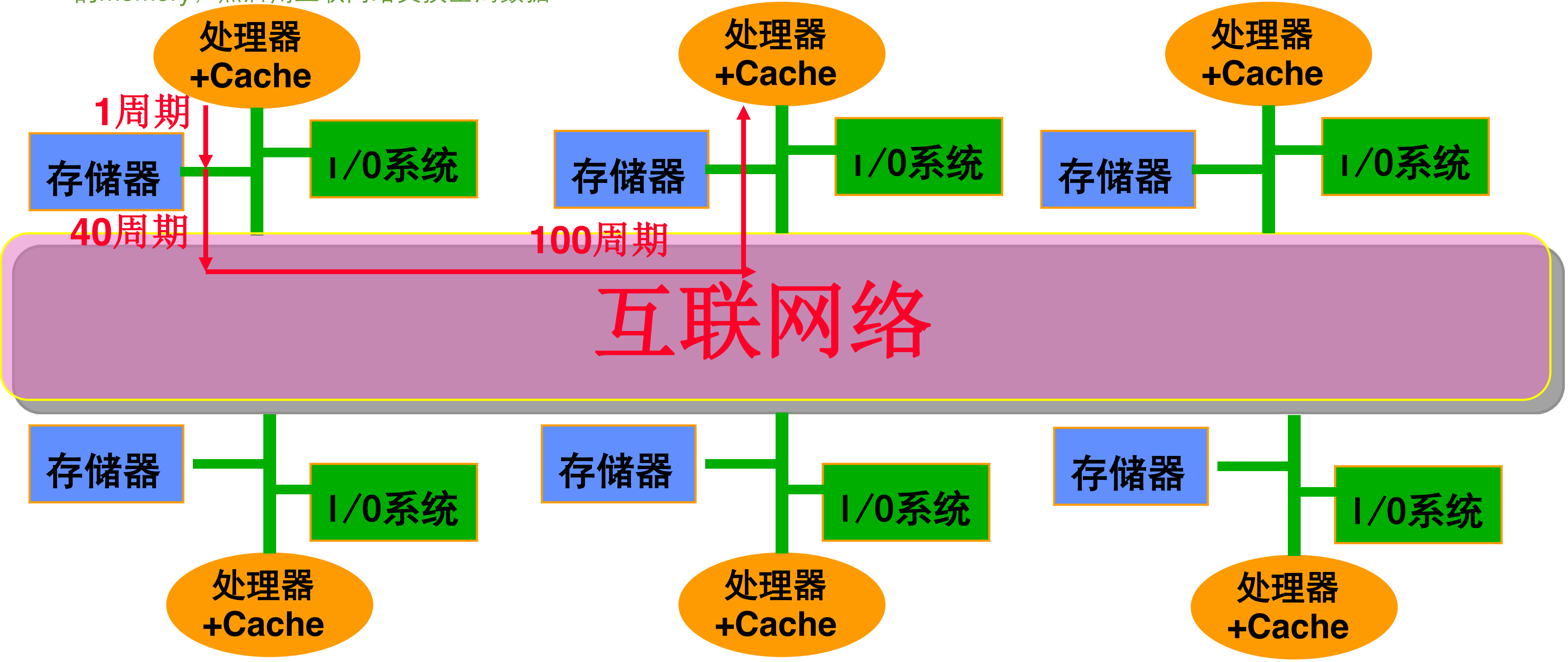
SMP互联

- 处理器连到存储器 并且 连到I/O
- 基于总线：所有的存储位置具有相同的访问时间，因而SMP = 对称多处理器（Symmetric MP）
  - 随着处理器和I/O的增加，共享限制带宽
- 交叉开关：扩展的成本很高
- 多级网络（与具有更高带宽的交叉开关相比，扩展的成本较低）
- “舞厅式”设计: 所有的处理器在网络的一侧，所有的存储器在网络的另一侧

大规模多处理器系统设计

- 注: 利用非统一访问时间（nonuniform access time ： numa）和可扩展的互联网络来实现分布（分布存储）

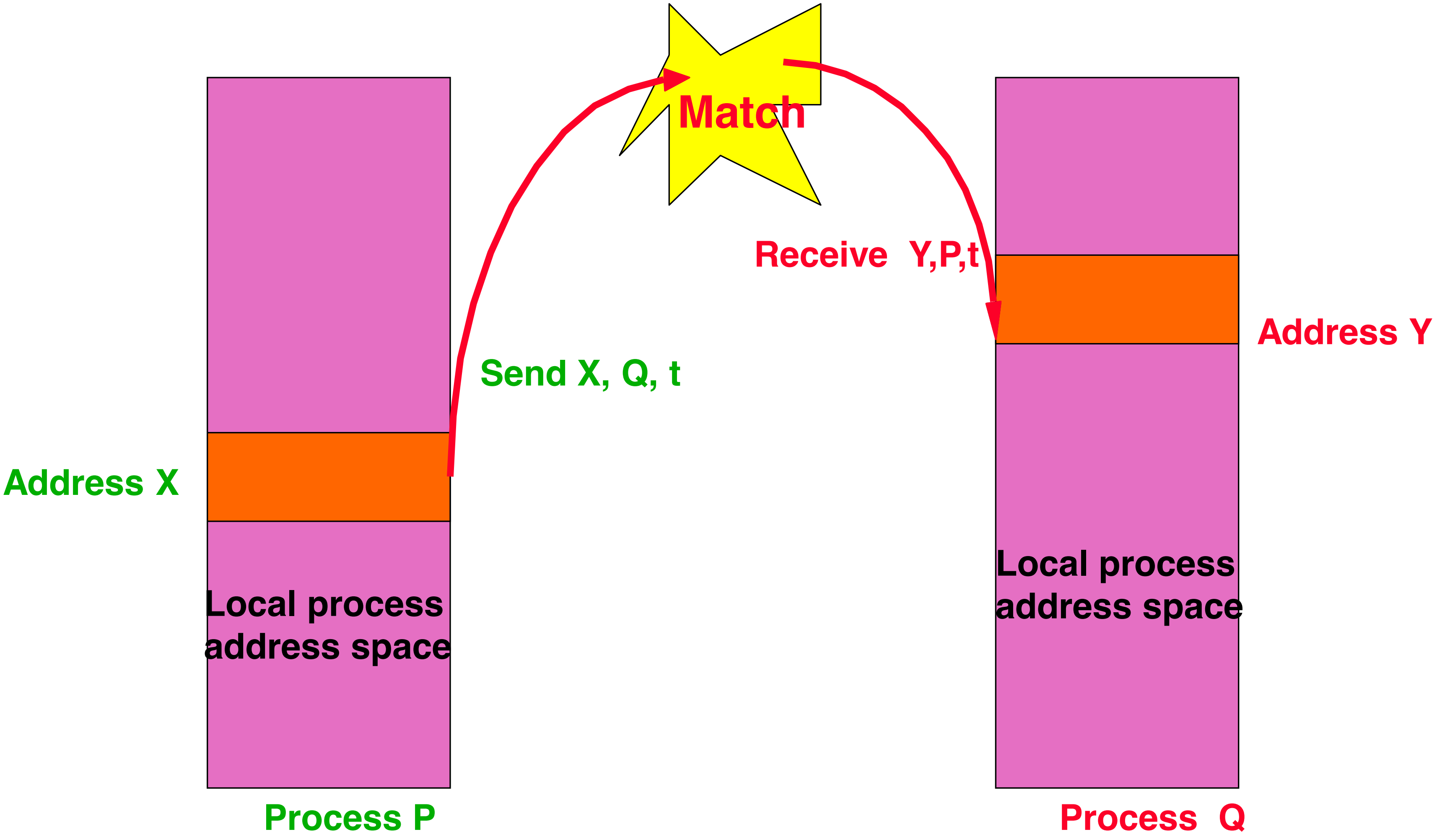
当互联的机器太多的时候，不管是bus还是crossbar还是共享内存，你想都不要想，你互联一万个机器两个足球场那么大，光走线得走多远，因此就不要谈uma了，只能做numa，让一个处理器自己带一个本地的memory，然后用互联网络交换全局数据



消息传递模型

- 所有计算机 (CPU、存储器、I/O设备)用显式I/O操作来完成通信
  - 本质上是NUMA，但利用I/O设备集成，而非存储系统
- Send 指定本地缓冲器 + 远程计算机的接收进程
- Receive 指定远程计算机的发送进程 + 存放数据的本地缓冲器
  - 通常，send包括进程标志（tag）并且receive遵从基于该标志的规则：单一匹配、任意匹配
  - 同步（Synch）: 当send完成、当缓冲器空闲、当请求接受（request accepted）、receive等待发送
- Send+receive => 存储器-存储器拷贝，每个原语都提供本地地址，并且 进行成对同步！

消息传递抽象


Match
Receive Y,P,t
Send X, Q, t
Address X
Address Y
Local process address space
Local process address space
Process P
Process Q

消息传递模型（续）

- Send+receive => 即使在单处理器上运行，也进行存储器-存储器拷贝，利用操作系统同步
- 信息传递的历史：
  - 由于只能发送数据给最临近的结点，因而网拓扑结构非常重要
  - 典型的同步：阻塞发送与接收
  - 后来，具有非阻塞发送的DMA，DMA负责将接收数据放在缓冲器中直到处理器真地开始接收，然后将数据传输到本地存储器
  - 后来，通过软件库来实现任意通信

通信模型

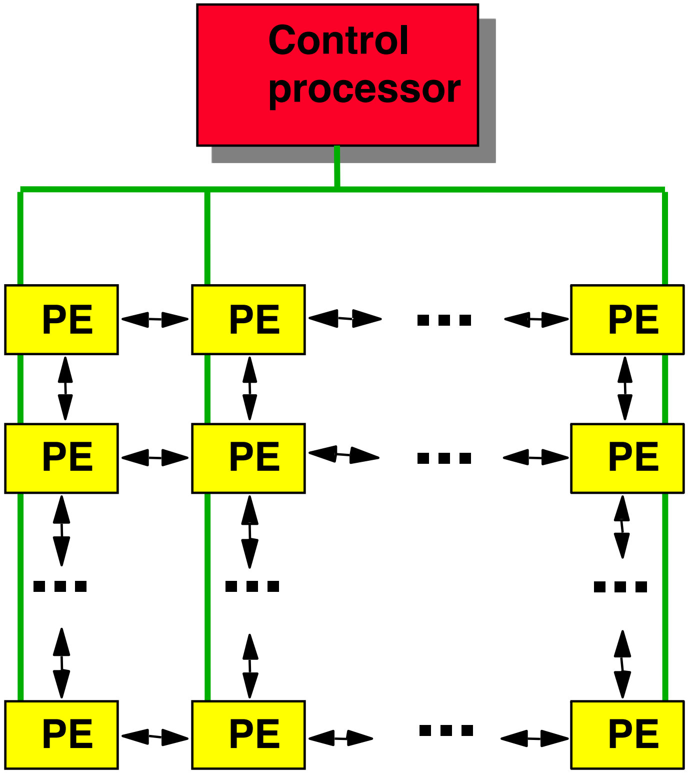
- 共享存储
  - 处理器通过共享地址空间进行通信
  - 易于在小规模机器上实现
  - 优点：
    - 单处理器和小规模多处理器系统选用的模型
    - 易于编程 (1) 与常用的集中式多处理机使用的通信机制兼容。
    - 低时延 (3) 通信开销较低，延时较小
    - 易于使用硬件控制的高速缓冲存储技术
- 消息传递
  - 处理器具有私用存储器，通过消息进行通信
  - 优点：
    - 使用硬件少，易于设计
    - 注意点在费时的 **非本地** 操作 通信是显示的，因此逼迫程序员不得不注意通信的过程和效率
- 在两种硬件的基础上可能支持两种软件模型

Flynn分类(1966)

- SISD (Single Instruction stream Single Data stream)
  - 单处理器
- SIMD (Single Instruction stream Multiple Data stream)
  - 分布存储SIMD (MPP, DAP, CM-1&2, Maspar)
  - 共享存储SIMD (STARAN, vector computers
    - 编程模型简单、低开销
- MIMD (Multiple Instruction stream Multiple Data stream)
  - 消息传递机器(Transputers, nCube, CM-5)
  - 非cache一致性的共享存储机器 (BBN Butterfly, T3D)
  - cache一致性的共享存储机器(Sequent, Sun Starfire, SGI Origin)
    - *使用商业化的微处理器*
- MISD (Multiple Instruction stream Single Data stream)
  - ???

数据并行模型

- 多个同样操作并行作用于一个大的规则数据结构（例如，数组）的每个元素
- 1个控制处理器广播到多个PEs
  - 当计算机很大时，需要分布完成多个重复PE的控制部分
- PE具有条件标志，因而可以实现调步
- 数据分布在每个存储器中
- 八十年代早期，VLSI => SIMD的复活：PE采用32个1位PE +片载存储器
- 数据并行编程语言给出数据在处理器上的布局



数据并行模型

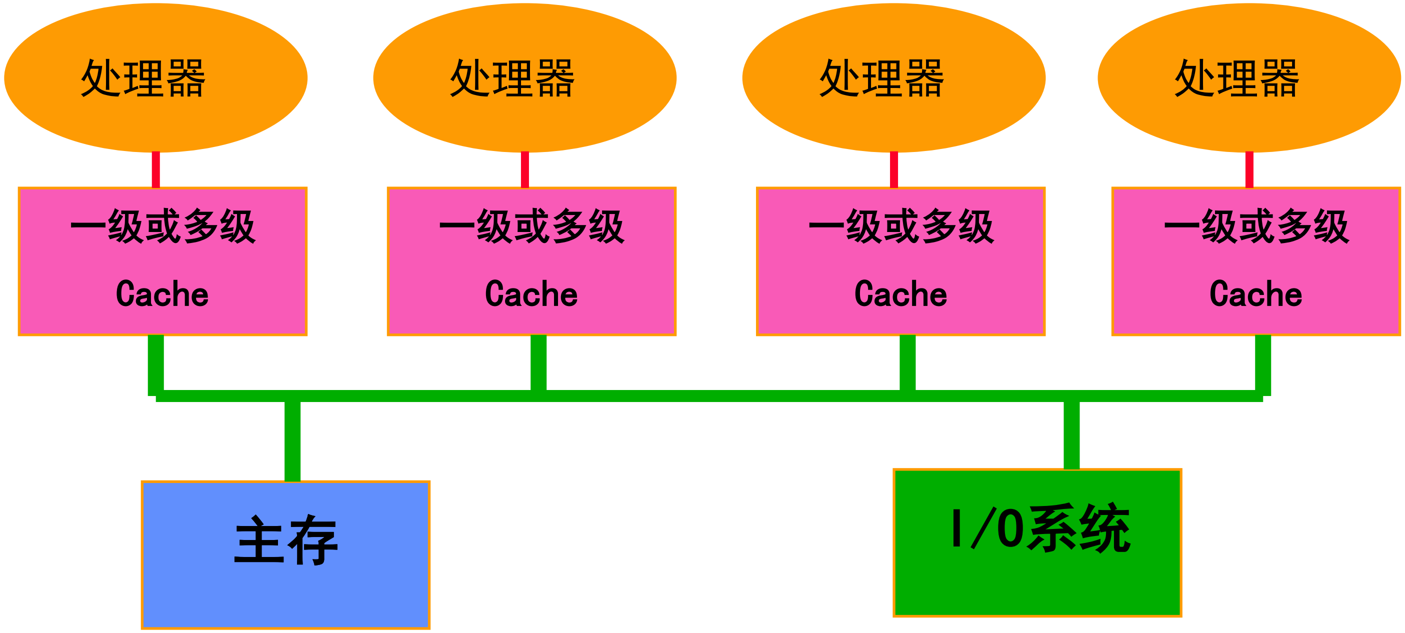
- 向量处理器具有类似的ISA，但是没有数据放置的限制
- SIMD产生的数据并行编程语言
- VLSI的发展产生了单片FPU和整个高速处理器(SIMD 的吸引力弱)
- SIMD编程模型发展为单程序多数据(SPMD) 模型
  - 所有的处理器执行同样的程序
  - 数据并行编程语言仍有用，立即完成所有的通信：大批同步（Bulk Synchronous）-- 一些在一个全局栅栏（ barrier ）之后完成所有通信的阶段

并行体系结构逐步集中

- 在通信帮助下，将完整计算机连接到一个可扩展网络（“闺房式”）
- 不同的编程模型对通信帮助的需求不同
  - 共享地址空间：与存储器紧密集成以捕获与其他处理器相互作用的存储器事件 + 以接受其他结点的请求
  - 消息传递：发送消息快速，并对到来消息响应：标志比较、分配缓冲器、传输数据、等待接收置入
  - 数据并行： 快速全局同步
- 高性能Fortran（HPF） 共享存储、数据并行；消息传递接口（MPI） 消息传递库；都可以在许多机器上工作，有多种不同实现

编译最不好做的事情就是到底数据该如何划分，而这件事恰恰是写程序的人知道的。因此只要你用户有效地把数据划分好，分到不同的地方后续编译就好做了。HPF就是这个思路。MPI的思路是：不管你机器是什么类型，你只要能把我的这些原语给实现好，MPI和C、C++、Fortran都可以结合，弄个库就好了

小规模多处理器系统



- **Cache的功效：**
  - 增加带宽 与 总线/存储器
  - 减少访问的时延
  - 对私有数据和共享数据都非常有效
- **cache一致性如何?**

这种体系结构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，而共享数据供多个处理器使用。共享数据主要是用来供处理器之间通过读写它们进行通信。私有数据缓冲在Cache中降低了平均访存时间和对存储器带宽的要求，使程序的行为类似于单机。共享数据可能会在多个Cache中被复制，这样做除了可降低访存时间和对存储器带宽的要求外还可减少多个处理器同时读共享数据所产生的冲突。但共享数据进入Cache也产生了一个新的问题，即Cache的一致性问题。

Cache一致性（coherence）问题

时间	事件	CPU A 的 Cache 内容	CPU B 的 Cache 内容	位置 X 的 存储器内容
0				1
1	A 读 X	1		1
2	B 读 X	1	1	1
3	A 将 0 存入 X	0	1	0

一致性的含义如何?

多机情况下，要保证每次读到的内容都是最新的写的结果
这种虽然不正式，但我们叫做strict consistency
这样的话所有的读写操作都需要序列化，于是只有读写之间的那些指令才能并行
就连读和读之间都不能随便交换次序了，因为这不是单一程序，我不知道别的程序在干啥。

- 非正式：
  - 任何读操作都必须返回最近写的内容
  - 太严格，也太难实现
  - 内存严格一致性(Strict Consistency)
- 2 different aspects of memory system behavior
  - Coherence: defines what values can be returned by a read
  - Consistency: determines when a written value will be returned by a read

coherence和consistency
coherence讲的是：对同一个位置，我写了读了这次读到底会读到啥值。
consistency讲的是：什么时候读回来的值就是我要的值，什么时候返回这个值这是可以切磋的

我们讲cache的coherence问题，讲memory的consistency问题

存储系统如何具有一致性（ Coherent）？

- 写序列化（write serialization） ：
  - 任何写操作的结果最终都会被任何一次读操作看见
  - 所有的写操作都被所有处理器以正确的次序看见
- A memory system is coherent if ：
  - 如果 P 写 x 后，P读 x，并其间没有其他对x的写操作，那么P读的都是P的写结果
  - 如果 P 写 x 后，P1 读 x， 且读和写之间足够远并其间没有其他对x的写操作，那么P的写结果将被P1看见
  - 对单一位置的写操作是序列化的：以一确定次序可见
    - 将看见最后的写
    - 否则将以不合逻辑的次序看见多次写（在较新的数值写后，还看见较旧的数值）

1任何写操作结果都必须被任何一次读操作看见，不能说我写了别人看不见（比如我写完有人立刻又写了我，我的写相当于白写了，这是不允许发生的）
2要保证所有的写操作都必须被所有的处理器以正确的次序看见

首先要work，不管时间多么长，首先要保证我写的最终你能读到。两种结构，SMP和大NUMA结构。SMP用窥视，NUMA用目录式
因为如果处理器太多那就没法用bus，也就没法用snoopy了

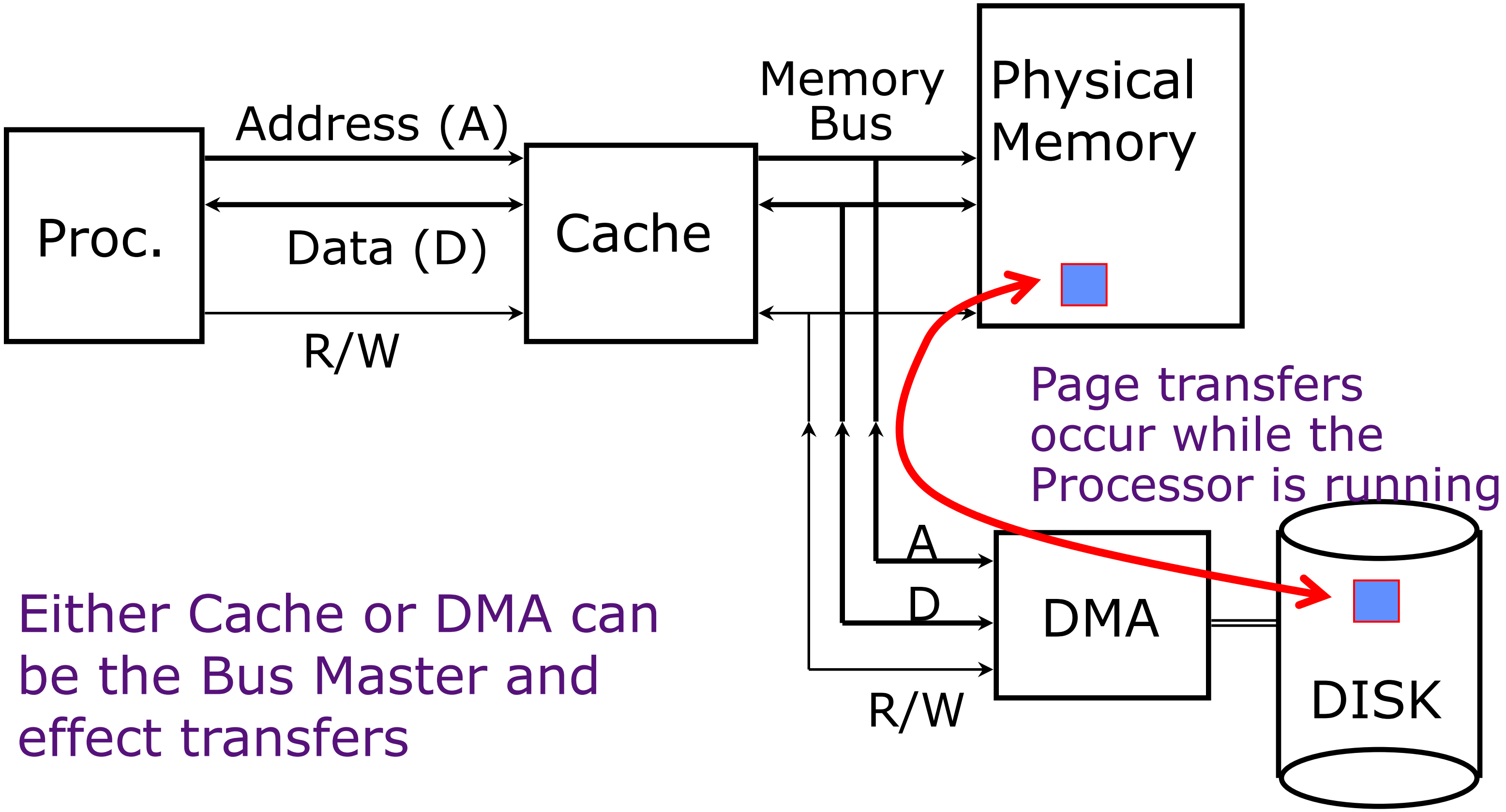
可能的硬件一致性解决方案

- Snooping Solution (Snoopy Bus):
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)
- Directory-Based Schemes

用一个目录表，在cache的每一行都有目录表，把信息记录下来，每做一次操作之后，通过message passing，将记录改为最新的状态。

  - Keep track of what is being shared in one centralized place
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Actually existed BEFORE Snooping-based schemes

Warmup: Parallel I/O

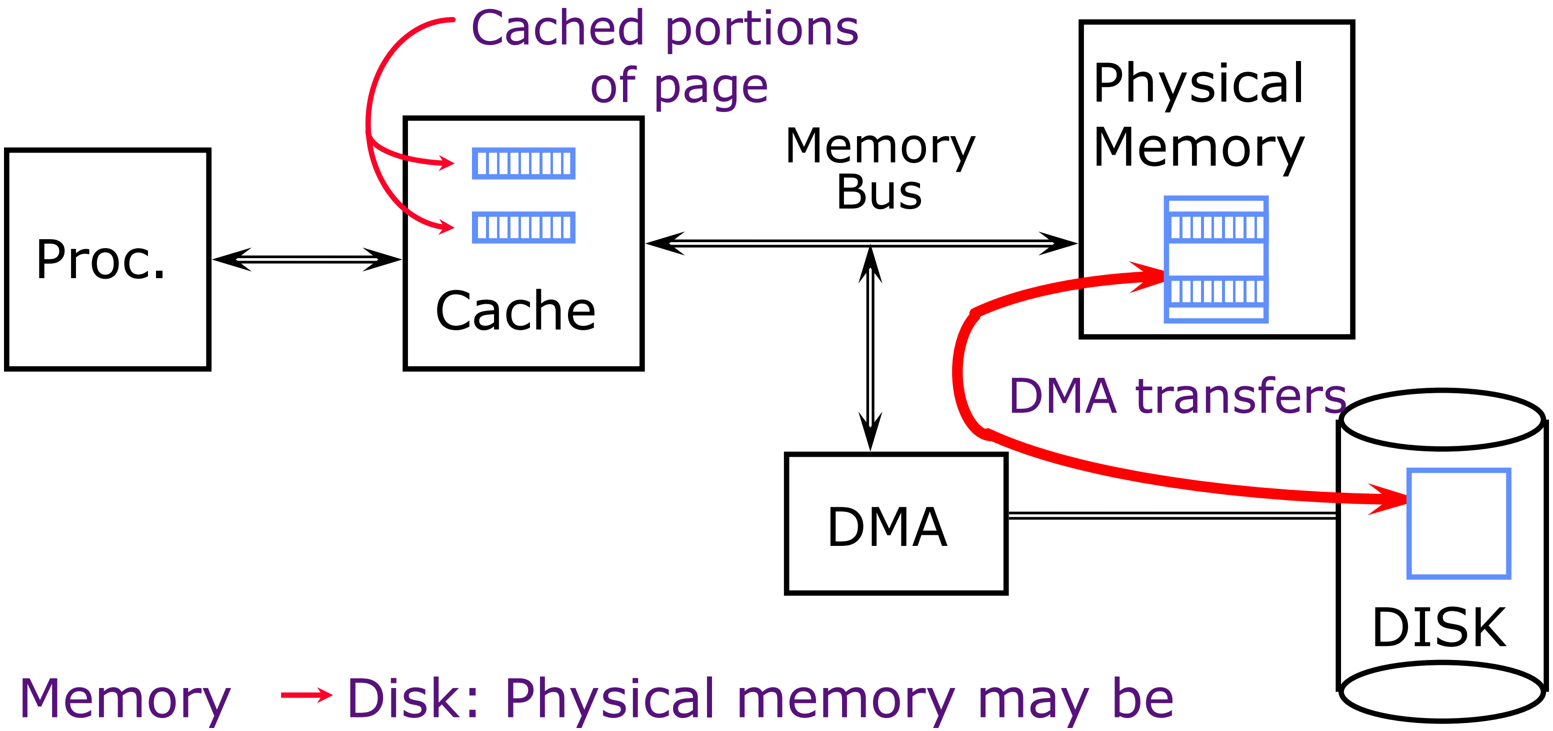


Page transfers occur while the Processor is running

Either Cache or DMA can be the Bus Master and effect transfers

(DMA stands for “Direct Memory Access”, means the I/O device can read/write memory autonomous from the CPU)

Problems with Parallel I/O

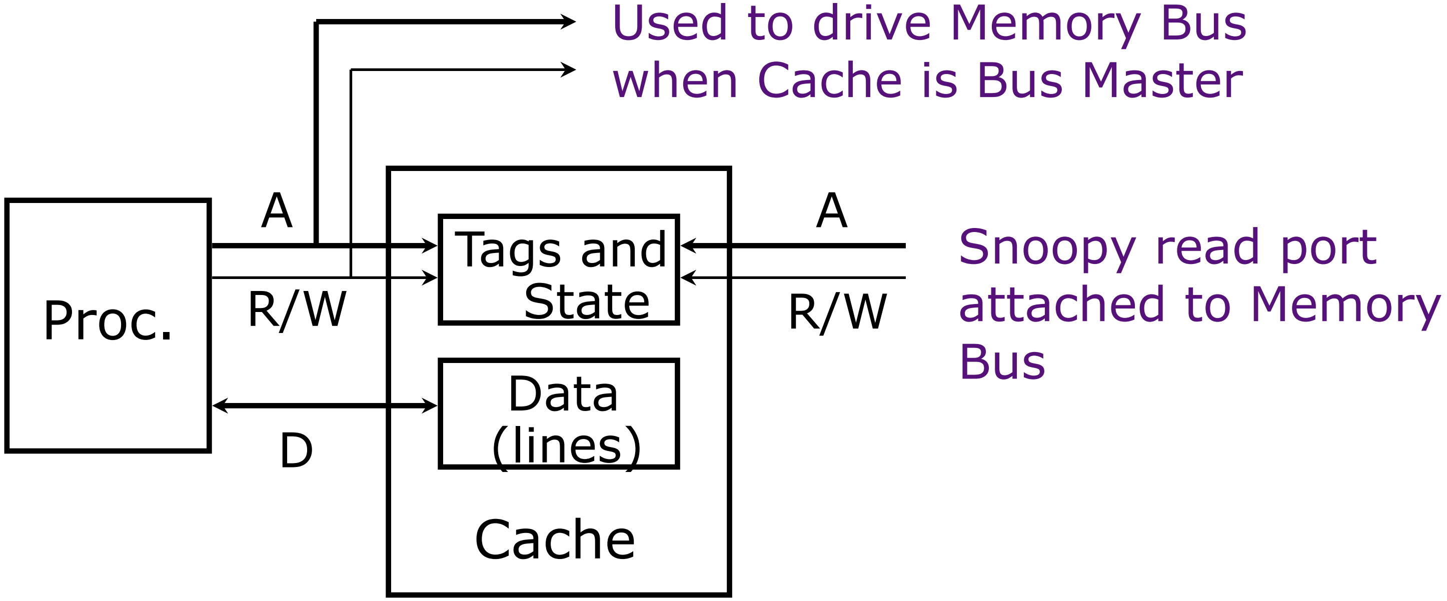


Memory → Disk: Physical memory may be stale if cache copy is dirty

Disk → Memory: Cache may hold stale data and not see memory writes

Snoopy Cache, *Goodman 1983*

- **Idea: Have cache watch (or snoop upon) DMA transfers, and then "do the right thing"**
- **Snoopy cache tags are dual-ported**


Used to drive Memory Bus when Cache is Bus Master
Proc.
A
R/W
D
Tags and State
Data (lines)
Cache
A
R/W
Snoopy read port attached to Memory Bus

Snoopy Cache Actions for DMA

Observed Bus Cycle	Cache State	Cache Action
DMA Read Memory → Disk	Address not cached	No action
	Cached, unmodified	No action
	Cached, modified	Cache intervenes
DMA Write Disk → Memory	Address not cached	No action
	Cached, unmodified	Cache purges its copy
	Cached, modified	???

基本窥探的协议

- Write Invalidate Protocol:
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and ***invalidate*** any copies
  - Read Miss:
    - Write-through: memory is always up-to-date
    - Write-back: snoop in caches to find most recent copy
- Write Broadcast Protocol (typically write through):
  - Write to shared data: broadcast on bus, processors snoop, and ***update*** any copies
  - Read miss: memory is always up-to-date
- Write serialization: bus serializes requests!
  - Bus is single point of arbitration

假设总线能够保证写序列化
这事儿不容易，但假设它能保证吧

窥探协议的一个例子

M的每个block和C的每个行都加标志表示其状态

- **Invalidation protocol, write-back cache**
- **Each block of memory is in one state:** 这三种状态包含了所有情况
  - **Clean in all caches and up-to-date in memory (Shared)**
  - **OR Dirty in exactly one cache (Exclusive)** 有且只有一个cache里头有脏数据 因为它脏的时候，必然广播给其它有这块数据的人，让它们都改成invalid了
  - **OR Not in any caches** 任何cache中都没有
- **Each cache block is in one state (track these):**
  - **Shared : block can be read** 我cache里有，主存里也有，有可能别的cache里也有
  - **OR Exclusive : cache has only copy, its writeable, and dirty** 当我变成exclusive的时候，我必然要发一个消息使得所有其它地方都变成invalid的，且使得内存里头变成exclusive
  - **OR Invalid : block contains no data** 这个cache就废了，没用了。我肯定没有，但别的至
- **Read misses: cause all caches to snoop bus**
  所有人都在snoopybus，这时候有人看到我read miss了，自然就有人把数据给我了
- **Writes to clean line are treated as misses**
  每次写了一次，我一定会变成exclusive了，把别人变成invalid了，当然别人如果本来是exclusive，那它必须先write back我再write back
  因为block是一块，虽然是同一块block，但可能我写的是一个部位你写的是另一个部位

Snoopy-Cache 状态机-I

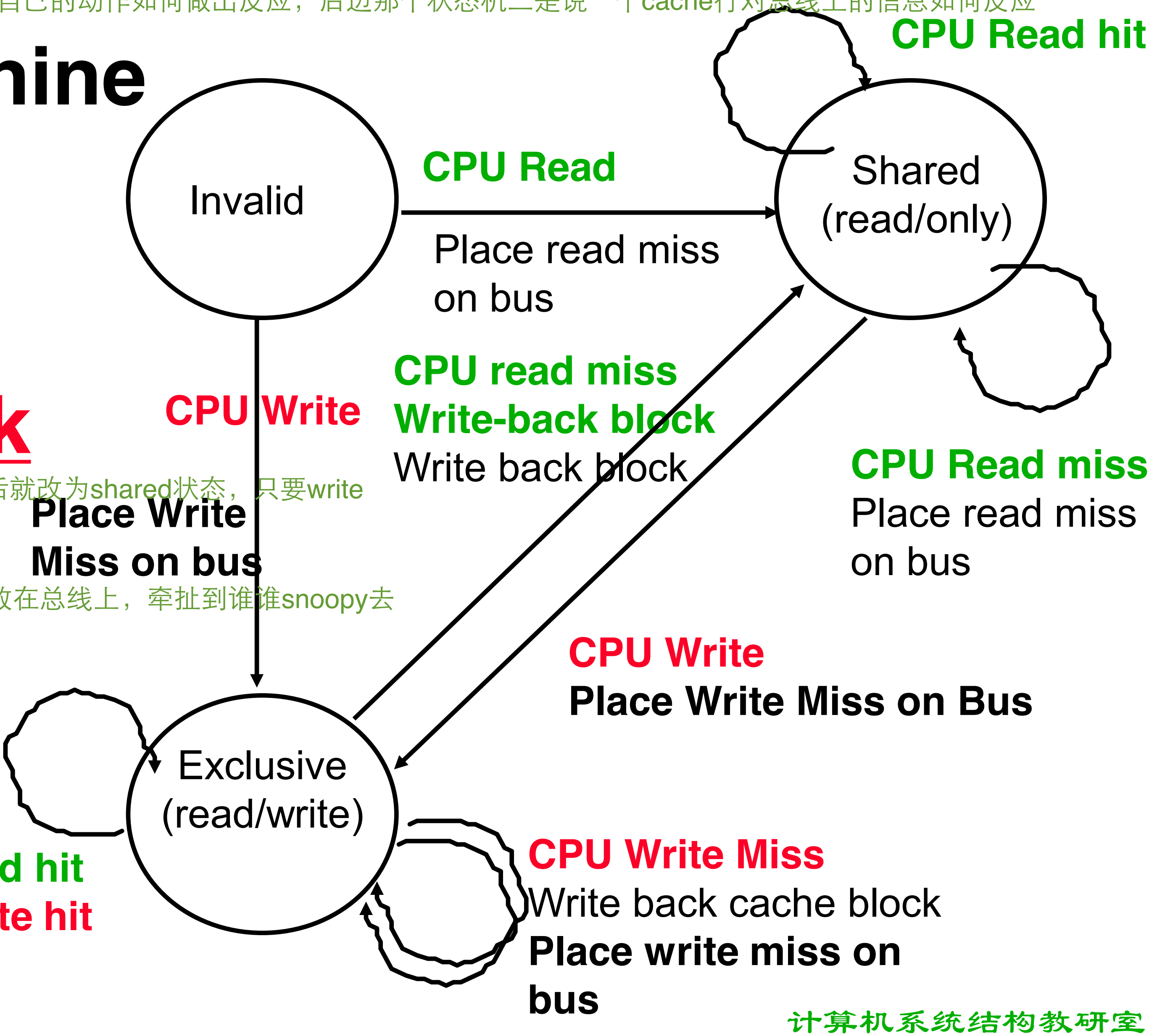
这个图是说一个cache行对自己的动作如何做出反应，后边那个状态机二是说一个cache行对总线上的信息如何反应

- **State machine for *CPU* requests for each cache block**

不管此时是何种状态，只要read了之后就改为shared状态，只要write了之后就改为exclusive状态。

注意write miss和read miss信号都是放在总线上，牵扯到谁谁snoopy去

Cache Block State



Invalid

Shared (read/only)

Exclusive (read/write)

CPU Read hit

CPU Read
Place read miss on bus

CPU Write
Place Write Miss on bus

CPU read miss
Write-back block
Write back block

CPU Read miss
Place read miss on bus

CPU Write
Place Write Miss on Bus

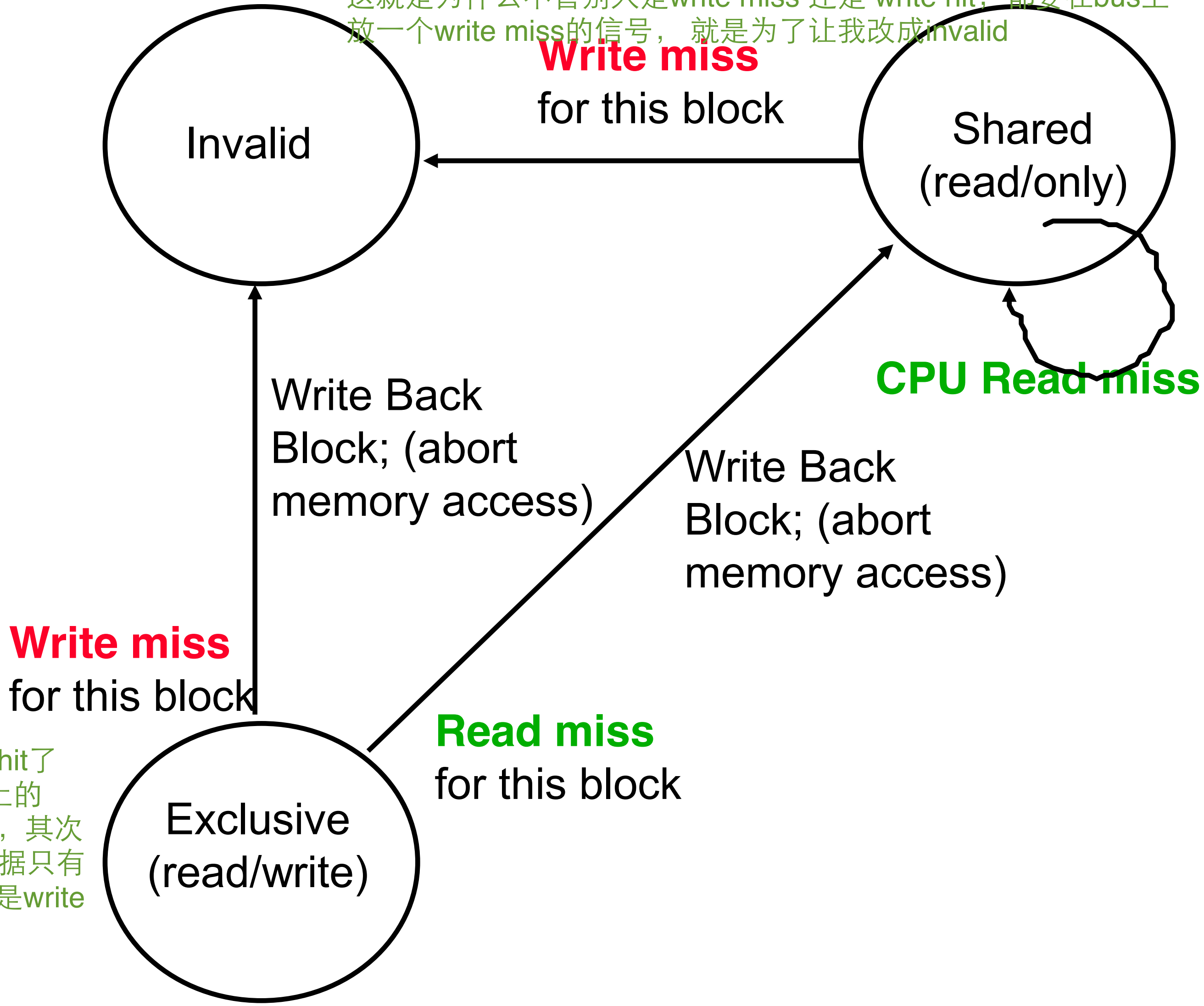
CPU read hit
CPU write hit

CPU Write Miss
Write back cache block
Place write miss on bus

Snoopy-Cache 状态机-II

- State machine for ***bus*** requests for each **cache block**

别人不管是write miss还是write hit了，我都得把自己改成invalid的
这就是为什么不管别人是write miss 还是 write hit，都要在bus上
放一个write miss的信号， 就是为了让我改成invalid



Invalid

Shared
(read/only)

Exclusive
(read/write)

Write miss
for this block

CPU Read miss

Write Back
Block; (abort
memory access)

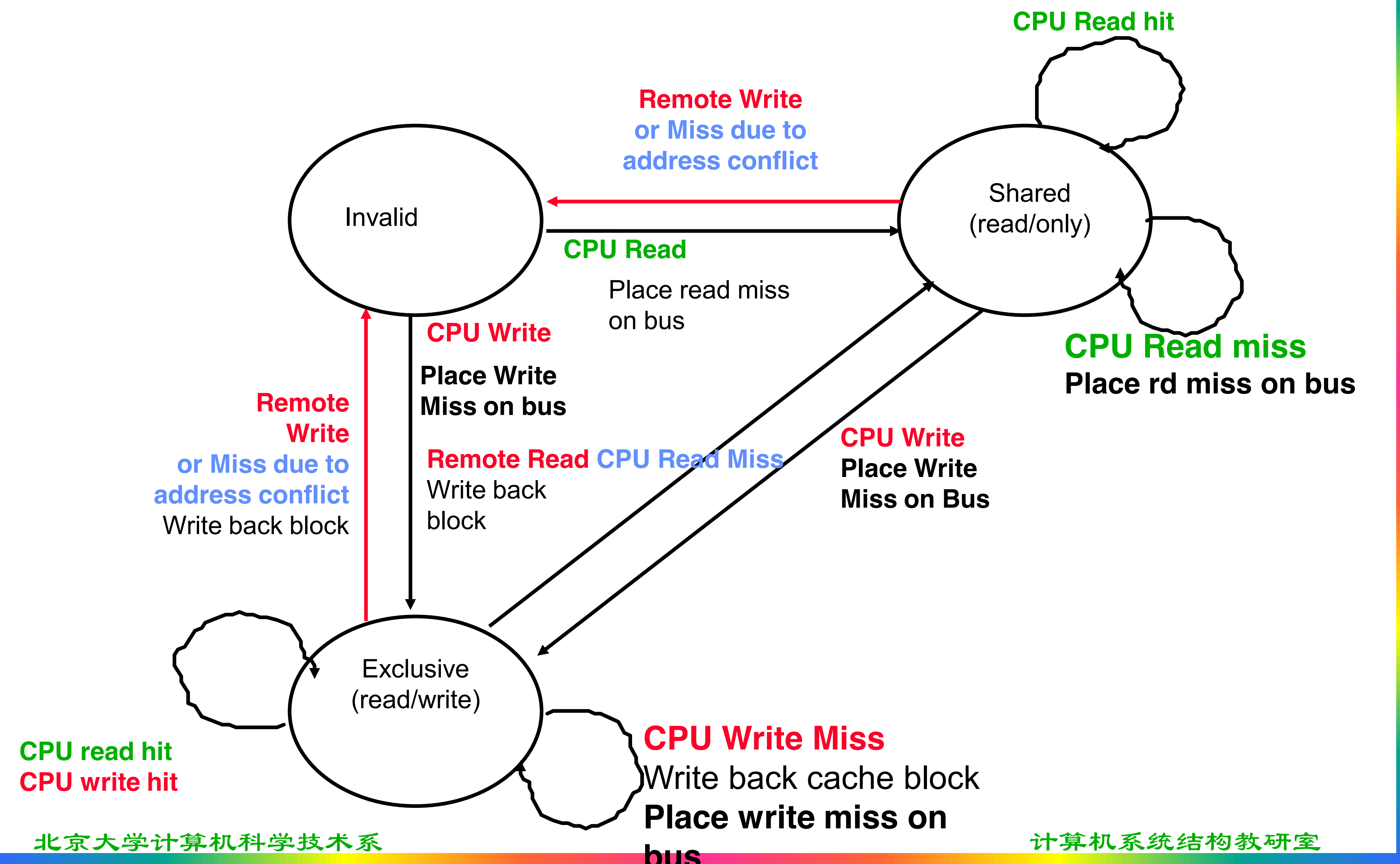
Write Back
Block; (abort
memory access)

Write miss
for this block

Read miss
for this block

首先，别人write和read如果hit了，是不会把hit信号放到bus上的
因此我听到的只有miss信号，其次当我是exclusive的时候，数据只有我这里独家一份，别人不管是write还是read都只可能是miss

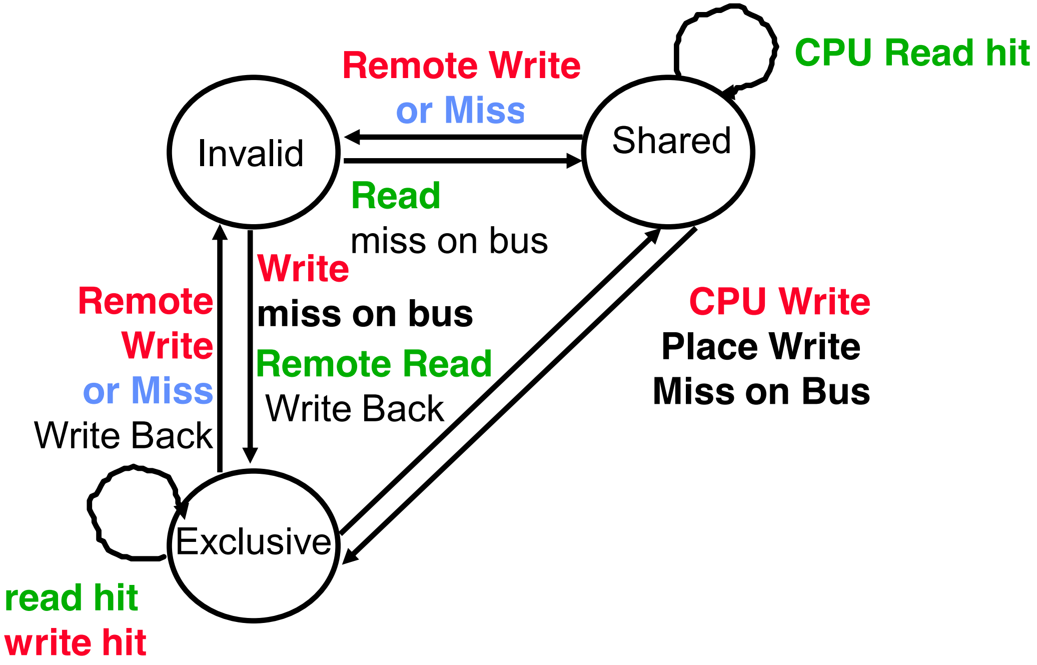
窥探Cache：状态机


CPU Read hit
Remote Write
or Miss due to
address conflict
Invalid
Shared
(read/only)
CPU Read
Place read miss
on bus
CPU Write
Place Write
Miss on bus
CPU Read miss
Place rd miss on bus
Remote
Write
or Miss due to
address conflict
Write back block
Remote Read CPU Read Miss
Write back
block
CPU Write
Place Write
Miss on Bus
Exclusive
(read/write)
CPU read hit
CPU write hit
CPU Write Miss
Write back cache block
Place write miss on
bus

示例

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

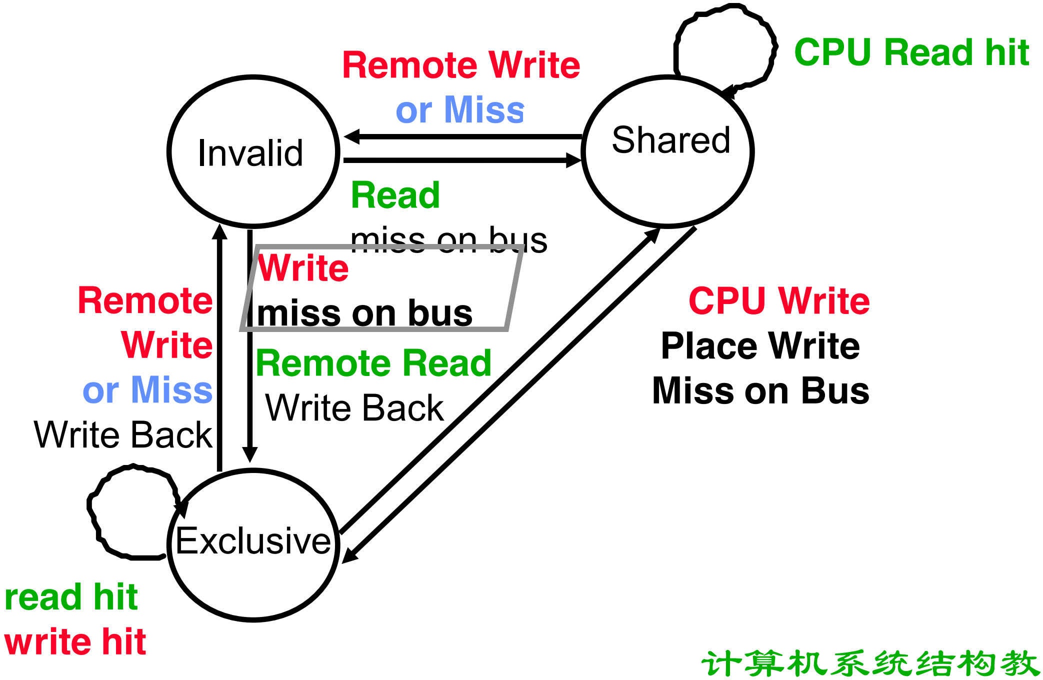
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2



示例（续一）

	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1: Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

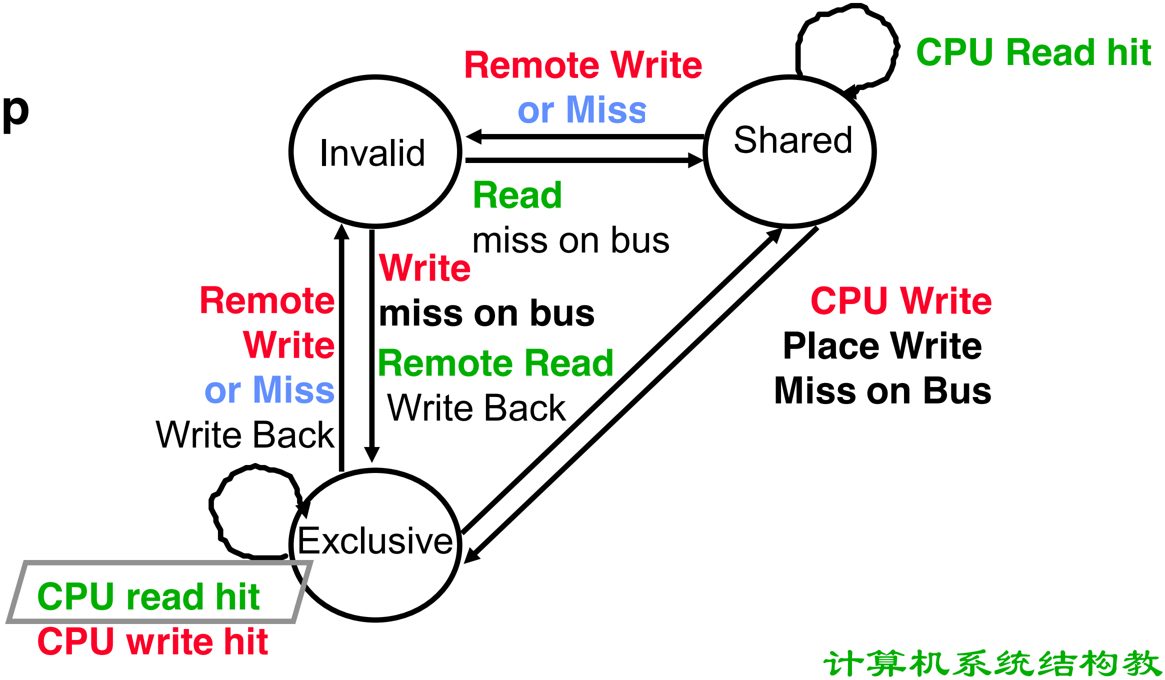
Assumes initial cache state is invalid and A1 and A2 map to same cache block,
but A1 ≠ A2.
Active arrow =



示例（续二）

	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1: Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

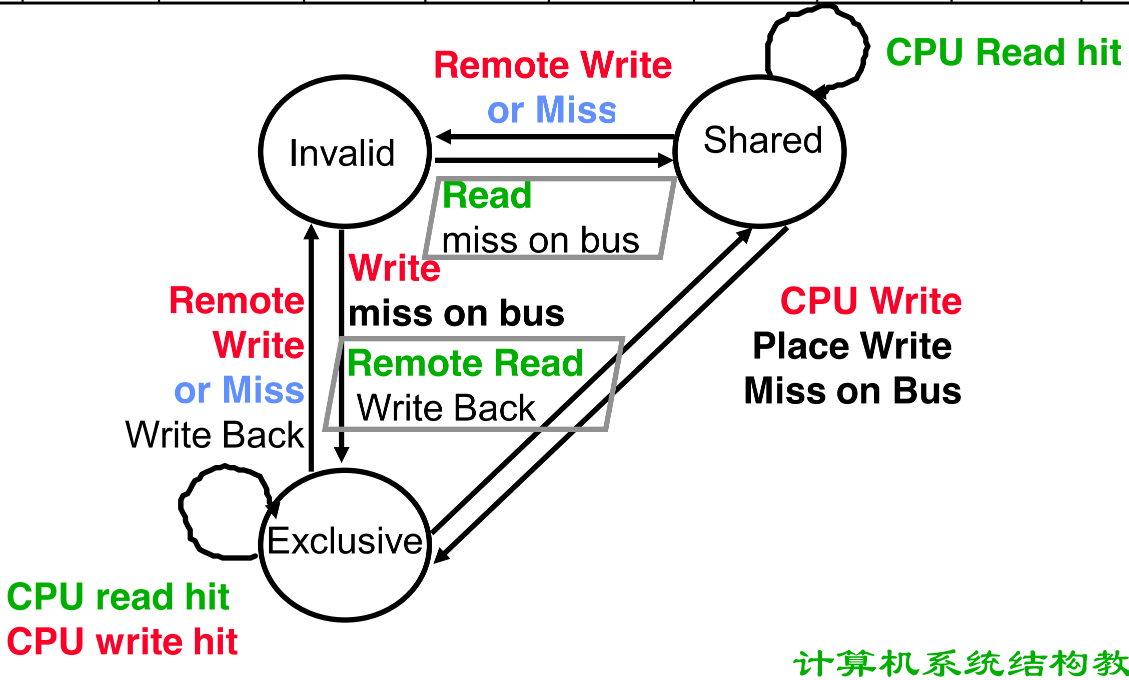
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2



示例（续三）

	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1: Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				*Shar.*	*A1*		*RdMs*	P2	A1			
	Shar.	A1	10				*WrBk*	P1	A1	10		*10*
				Shar.	A1	*10*	*RdDa*	P2	A1	10		10
P2: Write 20 to A1												10
P2: Write 40 to A2												10
												10

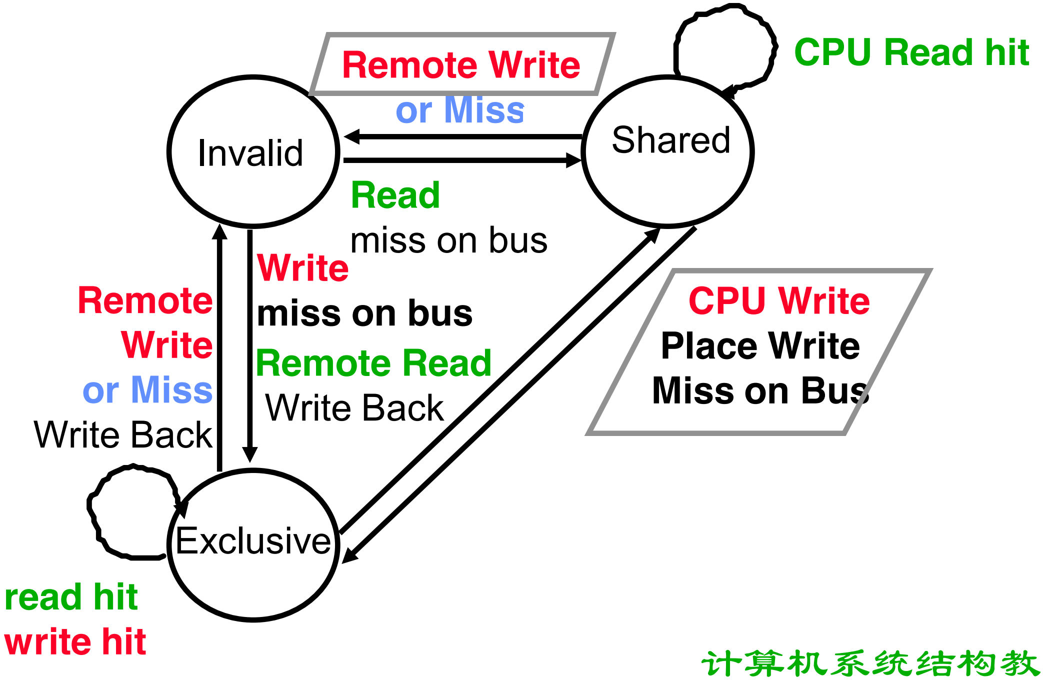
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2.



示例（续四）

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Valu	Actio	Proc.	Addr	Valu	Add	Valu
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10		10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1			10
P2: Write 40 to A2												10
												10

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2

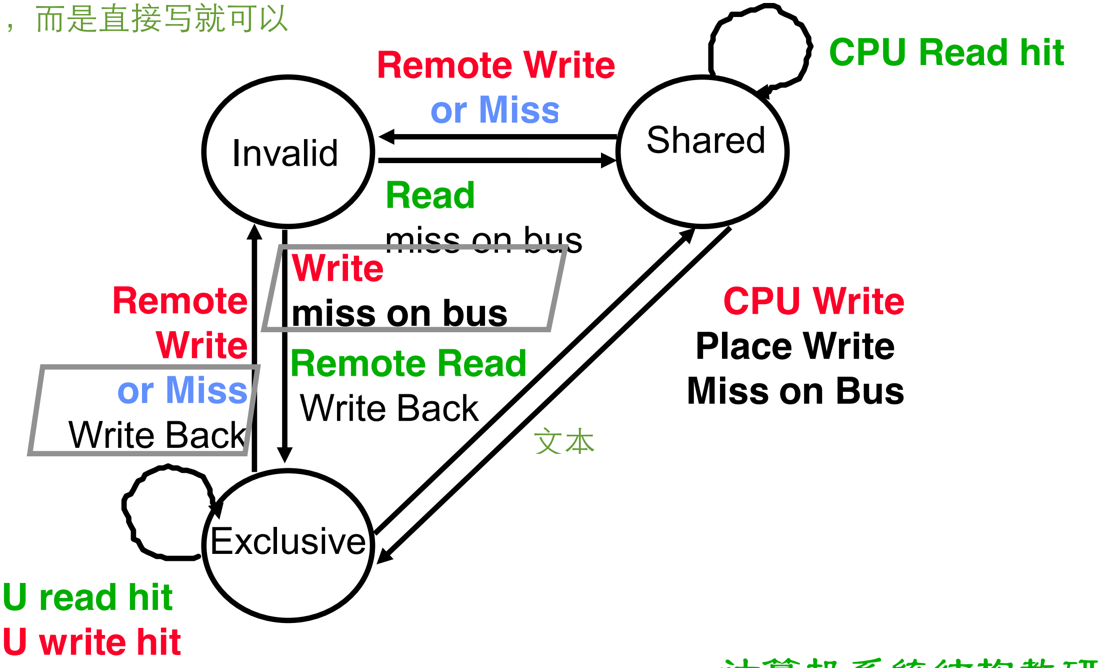


示例（续五）

	P1			P2			Bus				Memory	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10		10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1			10
P2: Write 40 to A2							WrMs	P2	A2			10
				Excl.	A2	40	WrBk	P2	A1	20		20

注意，最后一步，可以解释为P2写A2时世界上是write miss的，这种情况下就一定要先把cache行中的那个exclusive的数据给写回去
当然，如果时write hit的情况，就不需要先写回去了，而是直接写就可以

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2



实现复杂

- **Write Races:** 刚才假设总线是写序列化的。但让总线写序列化其实是挺难的
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - Bus transaction is not atomic: can have multiple outstanding transactions for a block
    - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    - Must track and prevent multiple misses for one block
- **Must support interventions and invalidations**

Snoopy Cache Coherence Protocols

write miss:

the address is *invalidated* in all other caches *before* the write is performed

read miss:

if a dirty copy is found in some cache, a write-back is performed before the memory is read

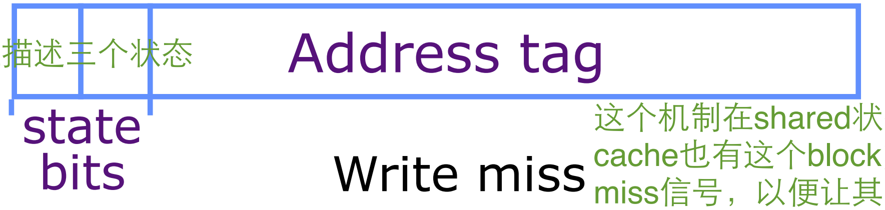
Cache State Transition Diagram

The MSI protocol

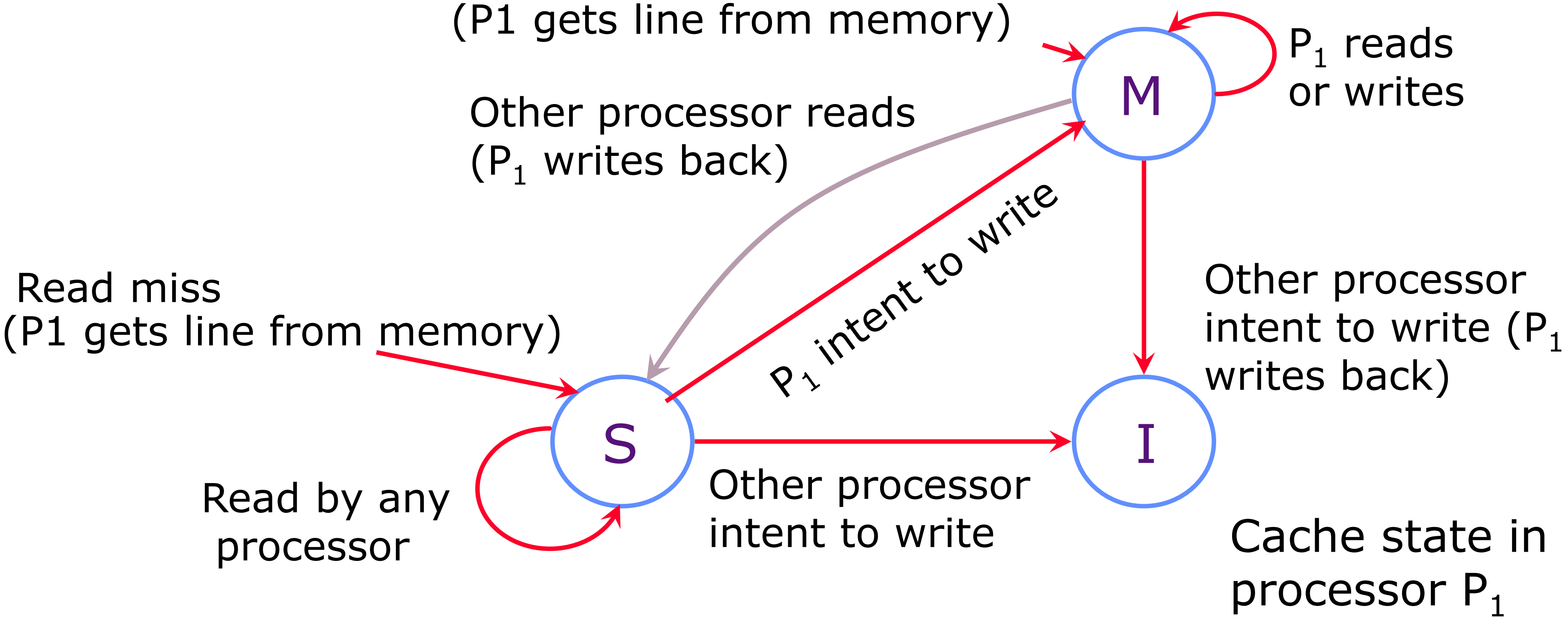
这个保证了对于一个主存块，多个处理器同时进行读写时我可以保证每个读到的都是最新的值，主存中每个block都维护这么一个状态机

Each cache line has state bits

M: Modified
S: Shared
I: Invalid



这个机制在shared状态时不能区分是只有我一个cache有这个block的内容还是其它cache也有这个block的内容，因此只要一个cache写了这个block，就要广播一个write miss信号，以便让其它所有的cache都改成invalid状态，这不好

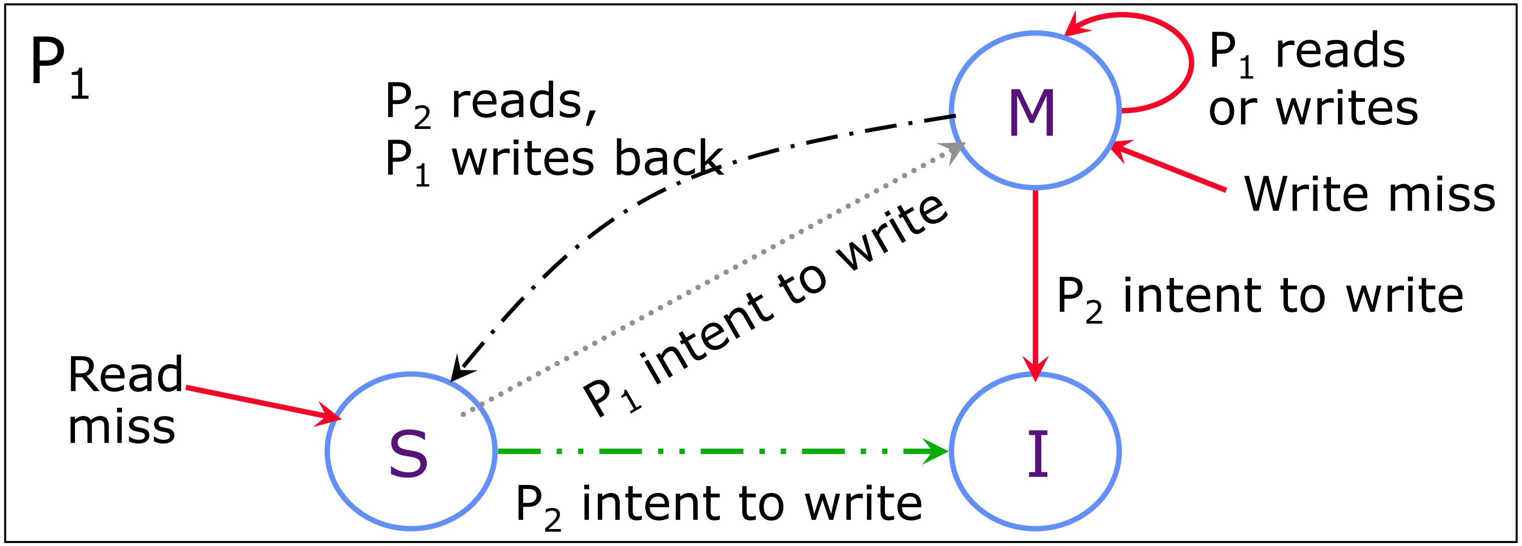


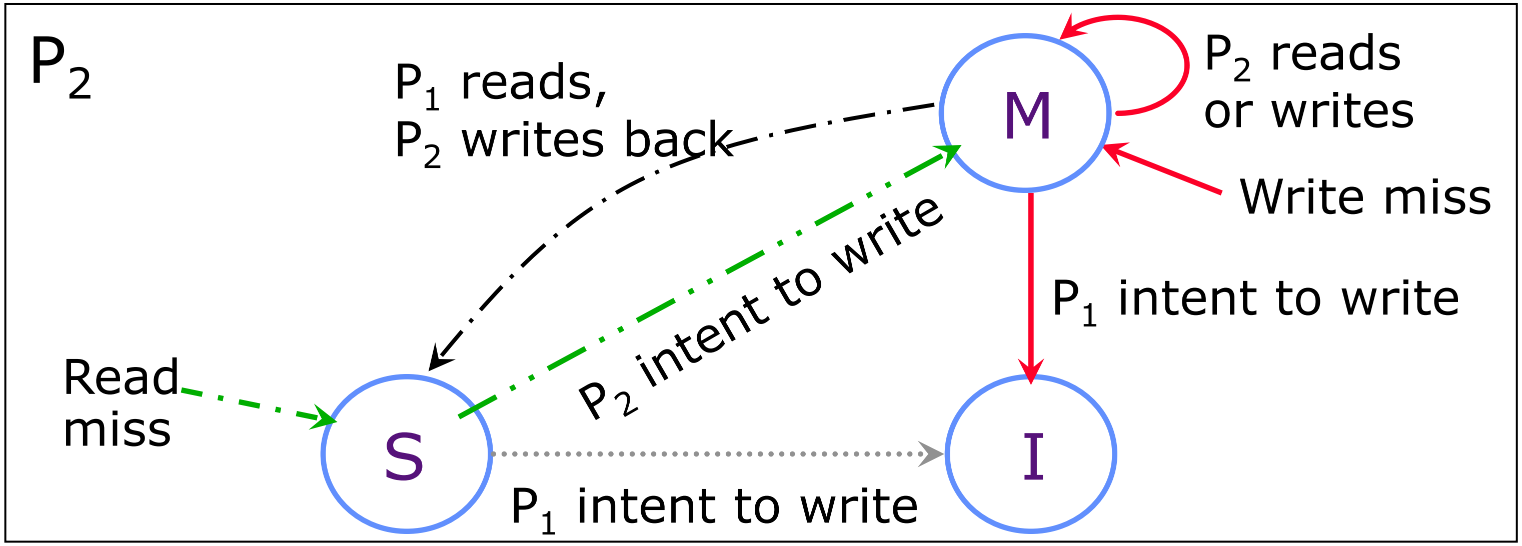
Write miss
(P1 gets line from memory)
M
P_1 reads or writes
Other processor reads
(P_1 writes back)
P_1 intent to write
Other processor intent to write (P_1 writes back)
Read miss
(P1 gets line from memory)
S
I
Read by any processor
Other processor intent to write
Cache state in processor P_1

Two Processor Example

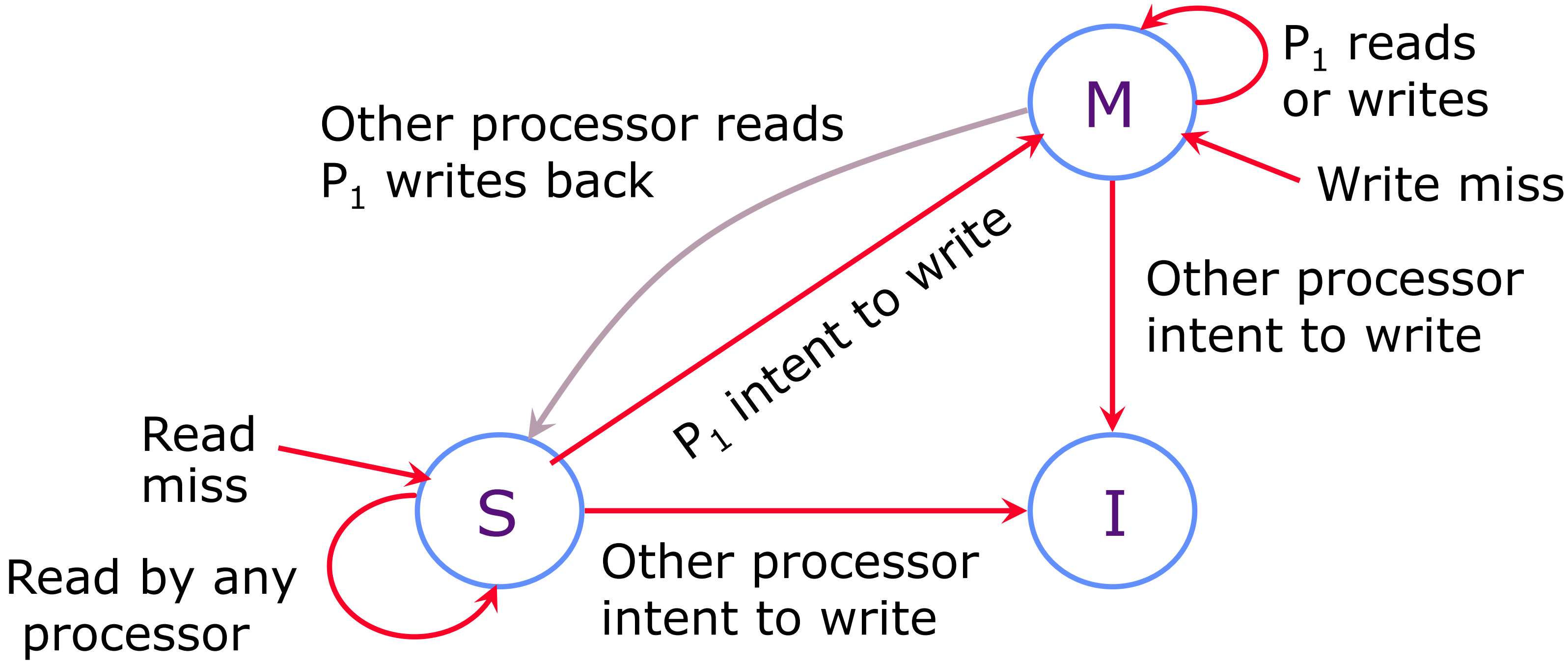
(Reading and writing the same cache line)

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





Observation


M
P1 reads or writes
Write miss
Other processor reads P1 writes back
P1 intent to write
Other processor intent to write
Read miss
S
Read by any processor
Other processor intent to write
I

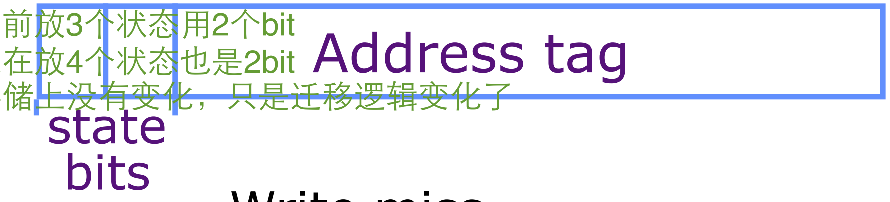
- If a line is in the **M** state then no other cache can have a copy of the line!
- Memory stays coherent, multiple differing copies cannot exist

MESI: An Enhanced MSI protocol

increased performance for private data

想描述处于某种状态时到底要不要回写,因此就把modified和exclusive状态分开,刚才三状态已经可以保证正确性，多分出来一个状态以免做过多多余动作。有可能经过定量分析，我们认为有个动作消耗太大，可能再分出来一个状态使得我们可以更好地避免消耗

Each cache line has a tag

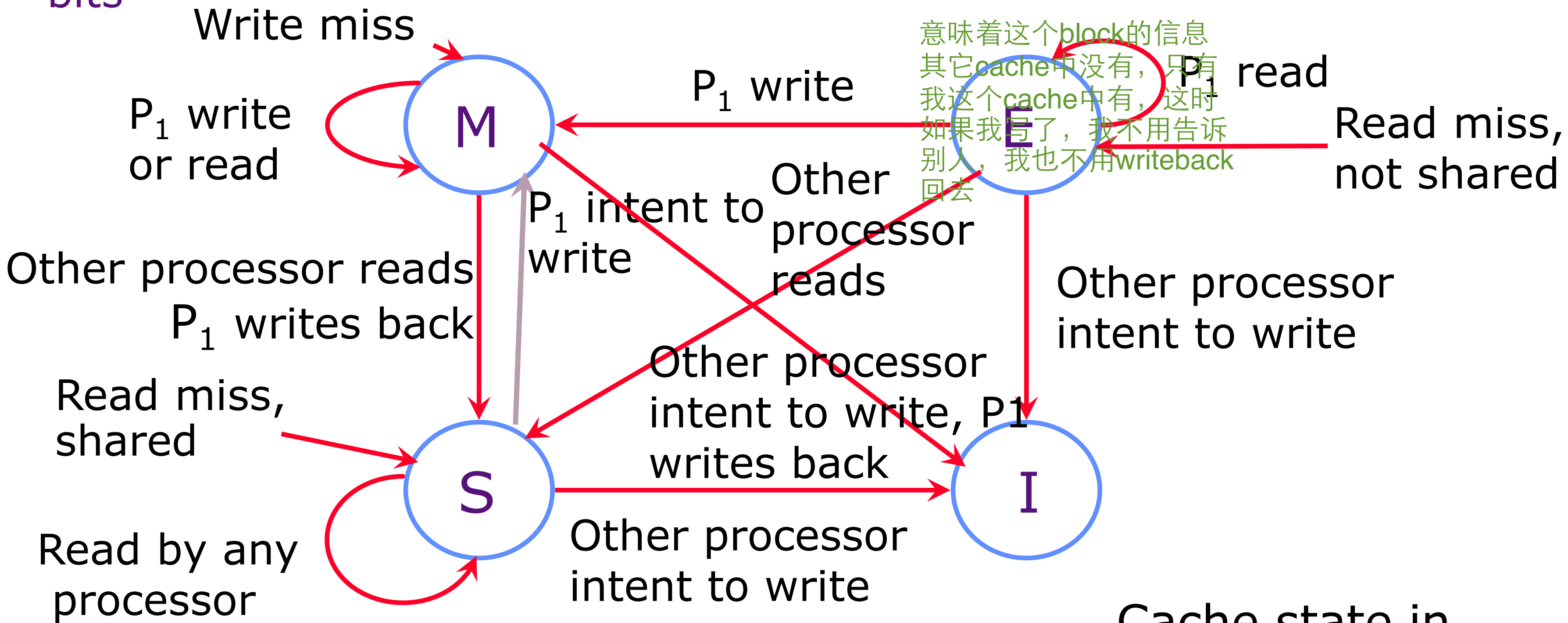


Address tag

state bits

以前放3个状态用2个bit
现在放4个状态也是2bit
存储上没有变化，只是迁移逻辑变化了

M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



Write miss

P_1 write or read

P_1 write

P_1 read

Read miss, not shared

Other processor reads P_1 writes back

P_1 intent to write

Other processor reads

Other processor intent to write

Other processor intent to write, P1 writes back

Read miss, shared

Read by any processor

Other processor intent to write

意味着这个block的信息其它cache中没有，只有我这个cache中有，这时如果我写了，我不用告诉别人，我也不用writeback回去

Cache state in processor P_1

Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of

1. Uniprocessor cache miss traffic
2. Traffic caused by communication
   - Results in invalidations and subsequent cache misses

- 4th C: *coherence miss*
  - Joins Compulsory, Capacity, Conflict

Coherency Misses

多处理器带来的，和之前讲的不同
一个cache一写，我的就无效了，是这种失

1. **True sharing misses** arise from the communication of data through the cache coherence mechanism
   - Invalidates due to 1st write to shared block
   - Reads by another CPU of modified block in different cache
   - Miss would still occur if block size were 1 word
2. **False sharing misses** when a block is invalidated because some word in the block, other than the one being read, is written into
   - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
   - Block is shared, but no word in block is actually shared
     ⇒ miss would not occur if block size were 1 word

true sharing miss：真的这个数据，就是一个a的值，给改掉了导致失效
false sharing miss：但是cache交换不会以byte为单位，可能不是改的a的值，但a所在的block失效了
若每个block只有一个byte，那就不会出现false的了，但为了空间局部性，必须要使得一个block包含好多byte
单一的处理器不存在这种东西

随着处理器数量增多，true sharing miss会增大，false的也会变大。那试试看增大cache会是什么情况呢？
一级cache不敢随便增大，因为太大会增加hit time，这会直接影响主频，因此我们希望增大二级、三级cache
发现增大cache只会降低以前学的几种失效，但true的和false的sharing miss几乎没有降低，因此核越多未必越快，
这两种多核带来的miss的影响是很可怕的事情，很难做好的

False Sharing

state	line addr	data0	data1	...	dataN

A cache line contains more than one word

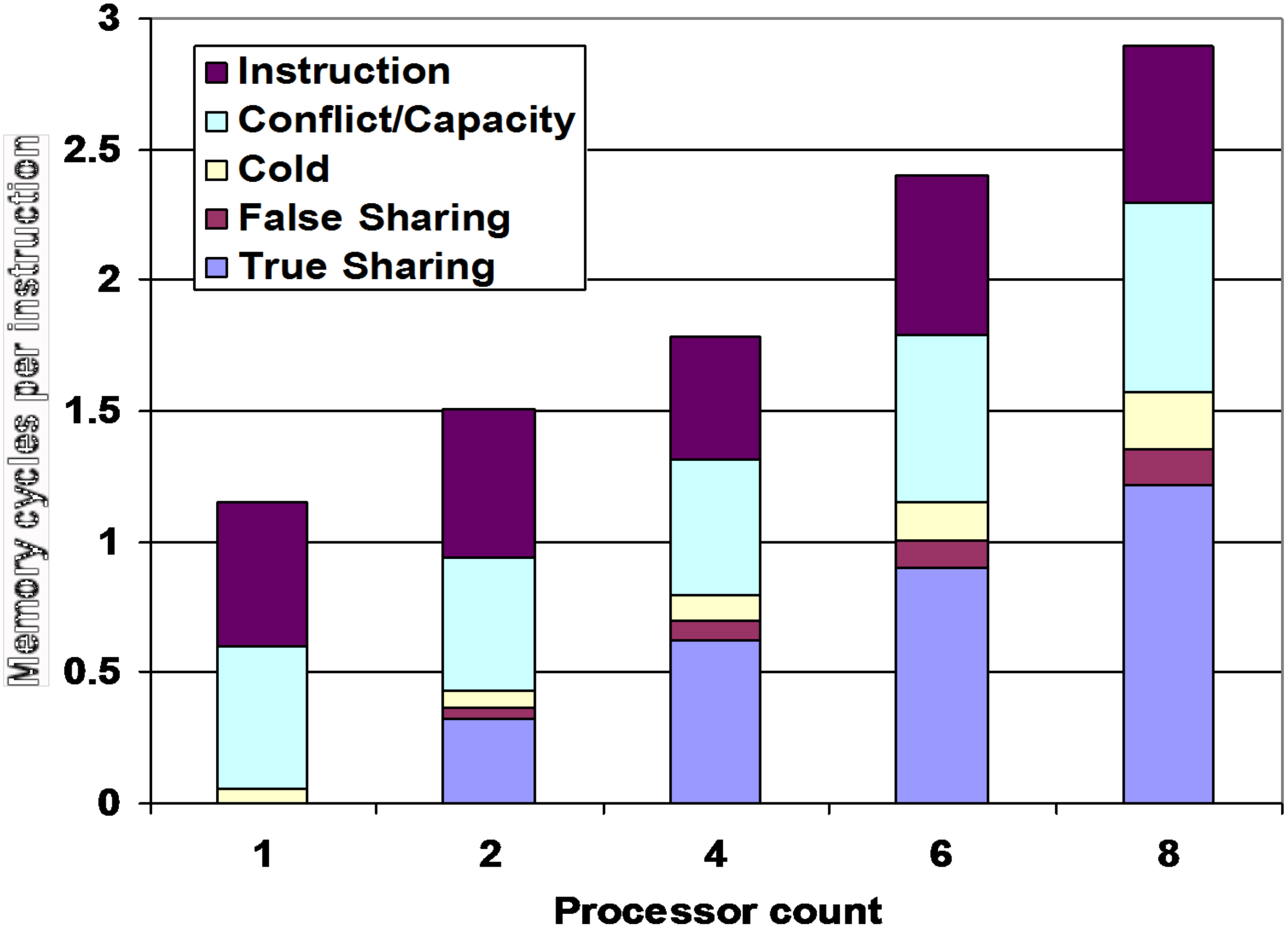
Cache-coherence is done at the line-level and not word-level

Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same line address.

What can happen?

MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs

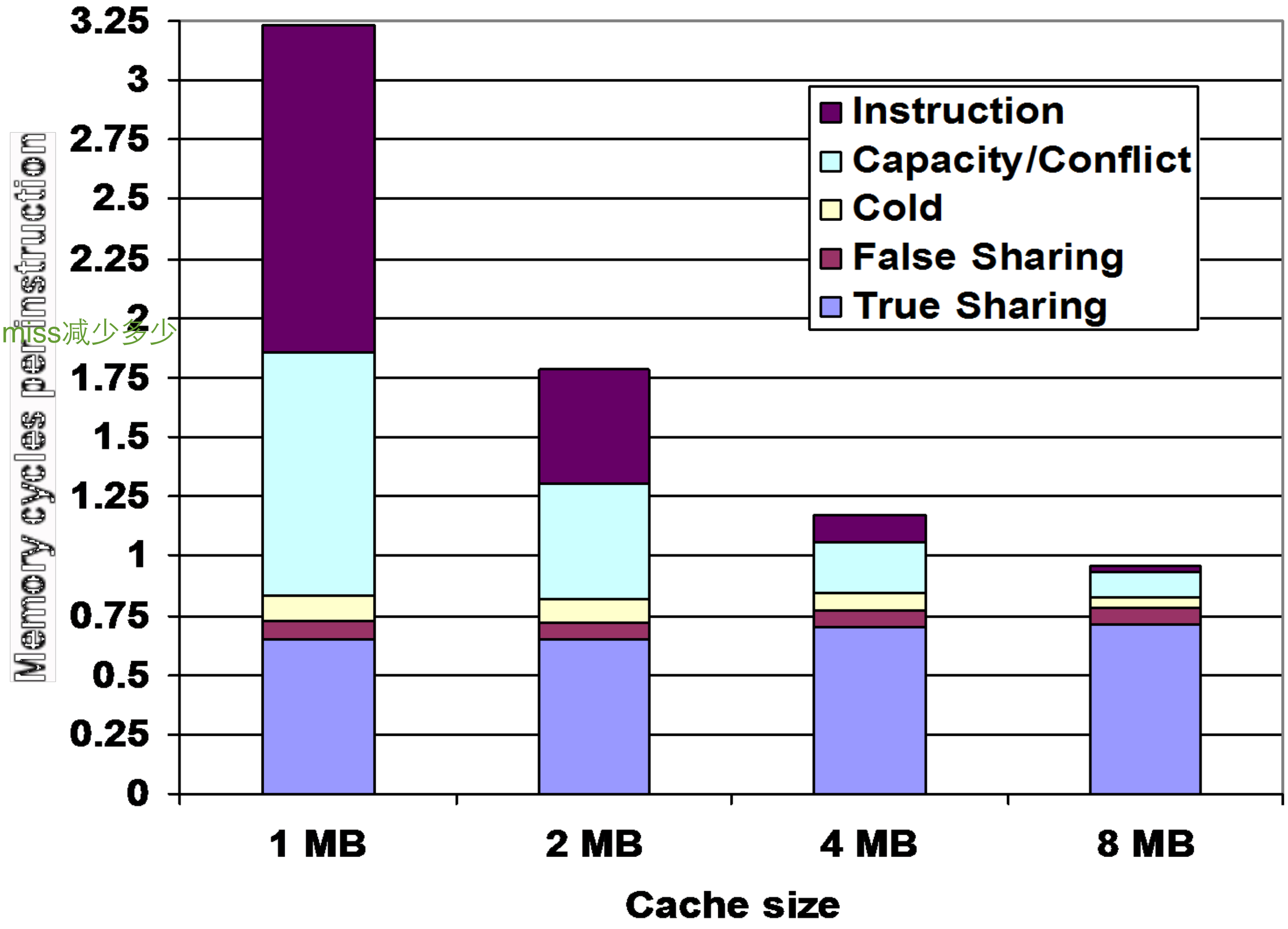


MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)

即增大cache容量，并不会导致coherence miss减少多少

- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)



较大的多处理器系统

- Separate Memory per Processor

当核过多的时候，就不敢在物理上让众多核共享一个物理主存了
就得让不同的核有不同的memory，通过互联网络连接起来

- Local or Remote access via memory controller
- 1 Cache Coherency solution: non-cached pages
- Alternative: directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is f(memory size) vs. f(cache size)

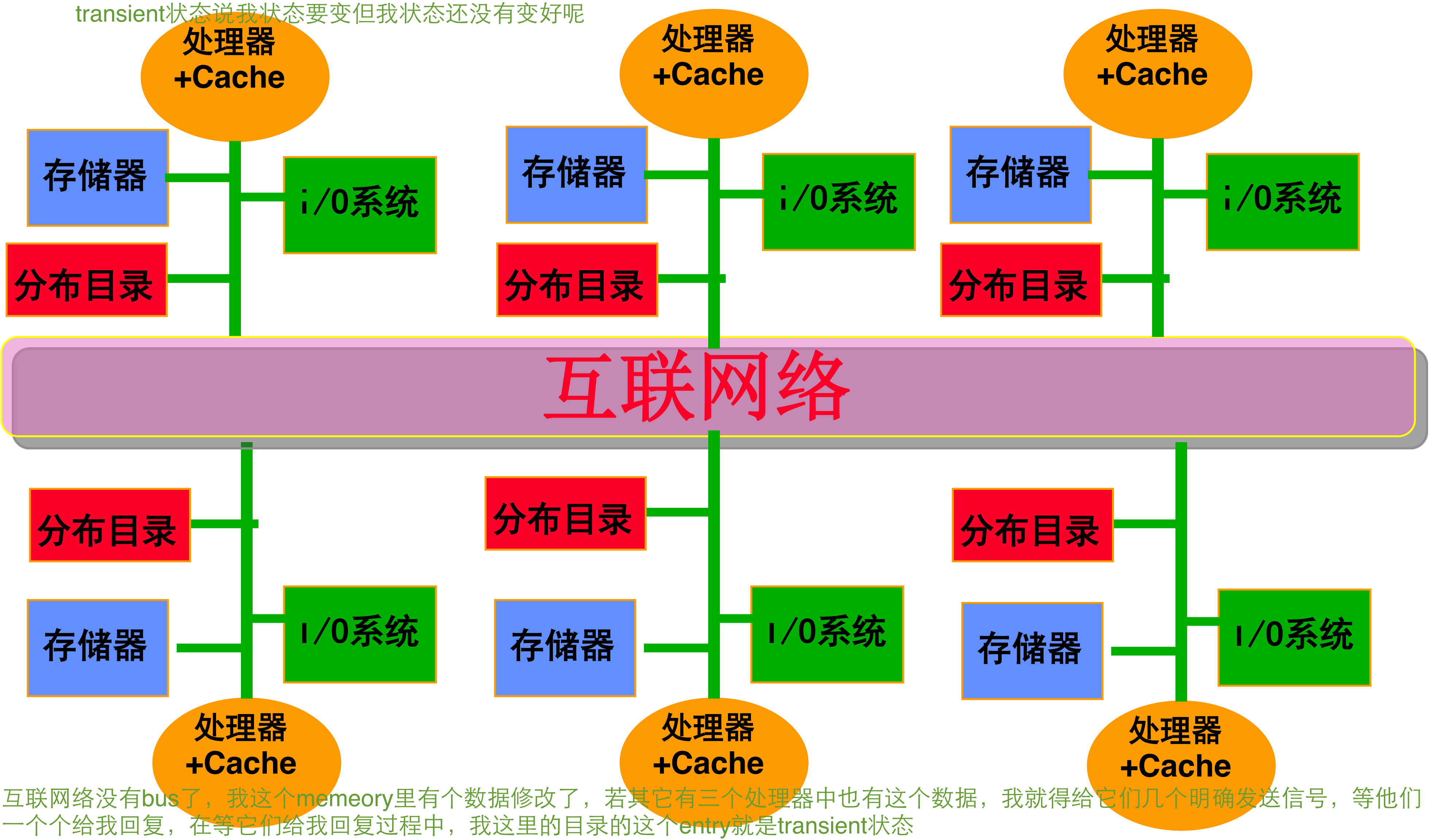
cache远小于主存大小，若主存每个block
都要放一个目录，那主存大部分是noncache
的，太浪费，目的是为了描述cache中放了
什么，因此就不再memory中放目录，在
cache中放目录就可以了。

- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which Processor have copies of their blocks

分布目录多处理器系统

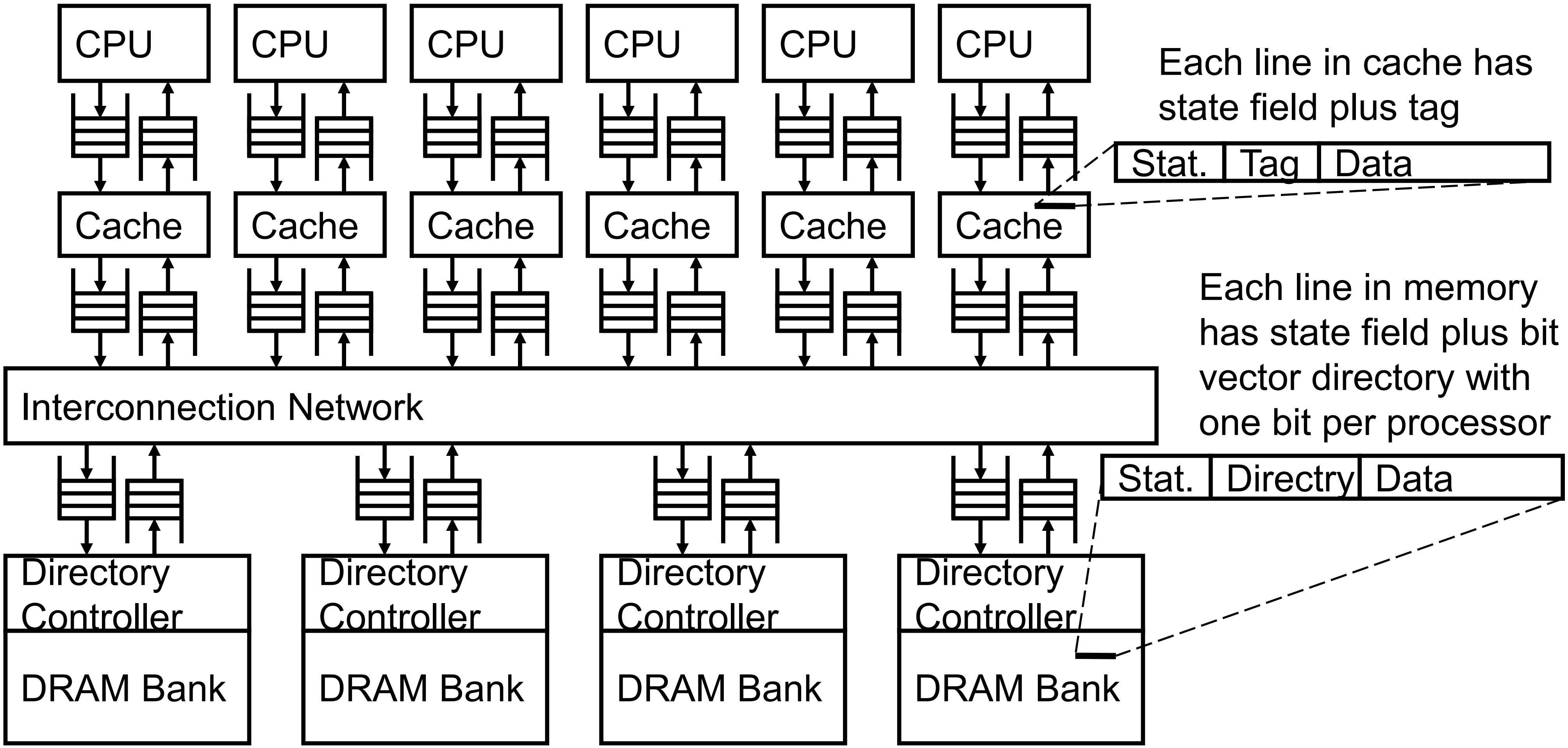
numa结构，每个处理器带一个存储器，但这些不同的存储器是统一编制的。这里说每个memory带个目录，但也可是实现成每个cache带一个目录

互联网假设是不可信的不可靠的，因此要TCP/IP协议，但我假设我的互联网络是质量非常高的，在这个网络上也保证是FIFO的，保证次序，谁先发谁先到谁先服务一切都是通过message send/reveive做的，发出去不知道要多少cycle能做完，因此要加一个transient状态说我状态要变但我状态还没有变好呢



互联网络没有bus了，我这个memeory里有个数据修改了，若其它有三个处理器中也有这个数据，我就得给它们几个明确发送信号，等他们一个个给我回复，在等它们给我回复过程中，我这里的目录的这个entry就是transient状态

Directory Cache Protocol



Each line in cache has state field plus tag

Each line in memory has state field plus bit vector directory with one bit per processor

- **Assumptions: Reliable network, FIFO message delivery between any given source-destination pair**

Cache States

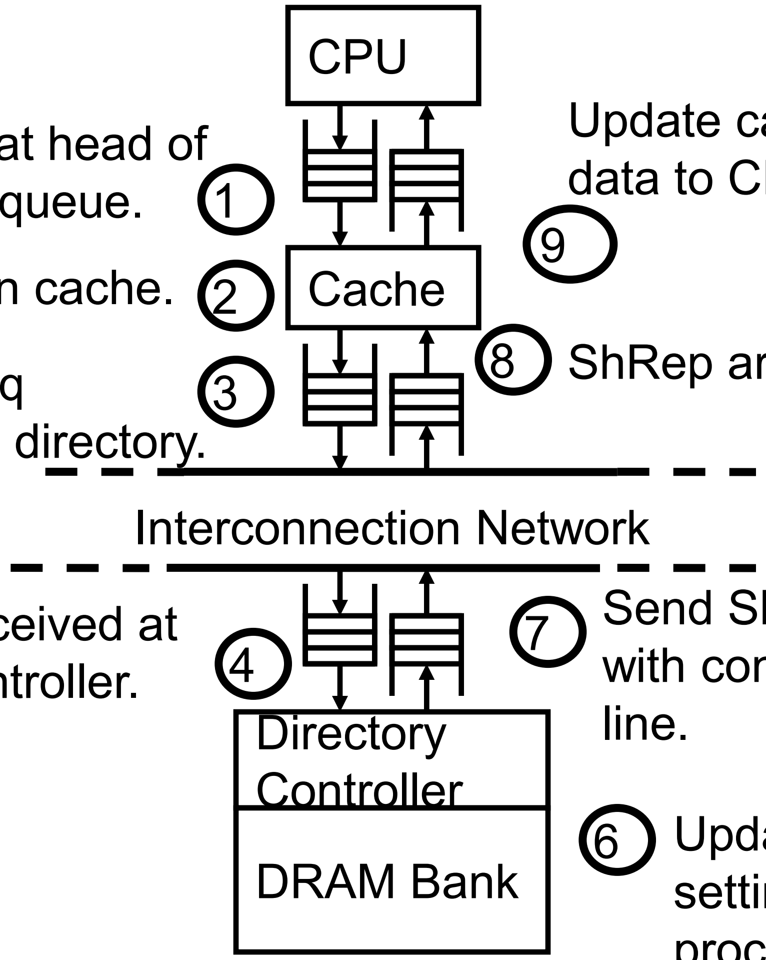
For each cache line, there are 4 possible states:

- C-invalid (= Nothing): The accessed data is not resident in the cache.
- C-shared (= Sh): The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
- C-modified (= Ex): The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
- C-transient (= Pending): The accessed data is in a *transient* state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

Home directory states

- For each memory line, there are 4 possible states:
  - R(dir): The memory line is shared by the sites specified in dir (dir is a set of sites). The data in memory is valid in this state. If dir is empty (i.e., dir = ε), the memory line is not cached by any site.
  - W(id): The memory line is exclusively cached at site id, and has been modified at that site. Memory does not have the most up-to-date data.
  - TR(dir): The memory line is in a transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.
  - TW(id): The memory line is in a transient state waiting for a line exclusively cached at site id (i.e., in C-modified state) to make the memory line at the home site up-to-date.

Read miss, to uncached or shared line



1 Load request at head of CPU->Cache queue.

2 Load misses in cache.

3 Send ShReq message to directory.

4 Message received at directory controller.

5 Access state and directory for line. Line’s state is R, with zero or more sharers.

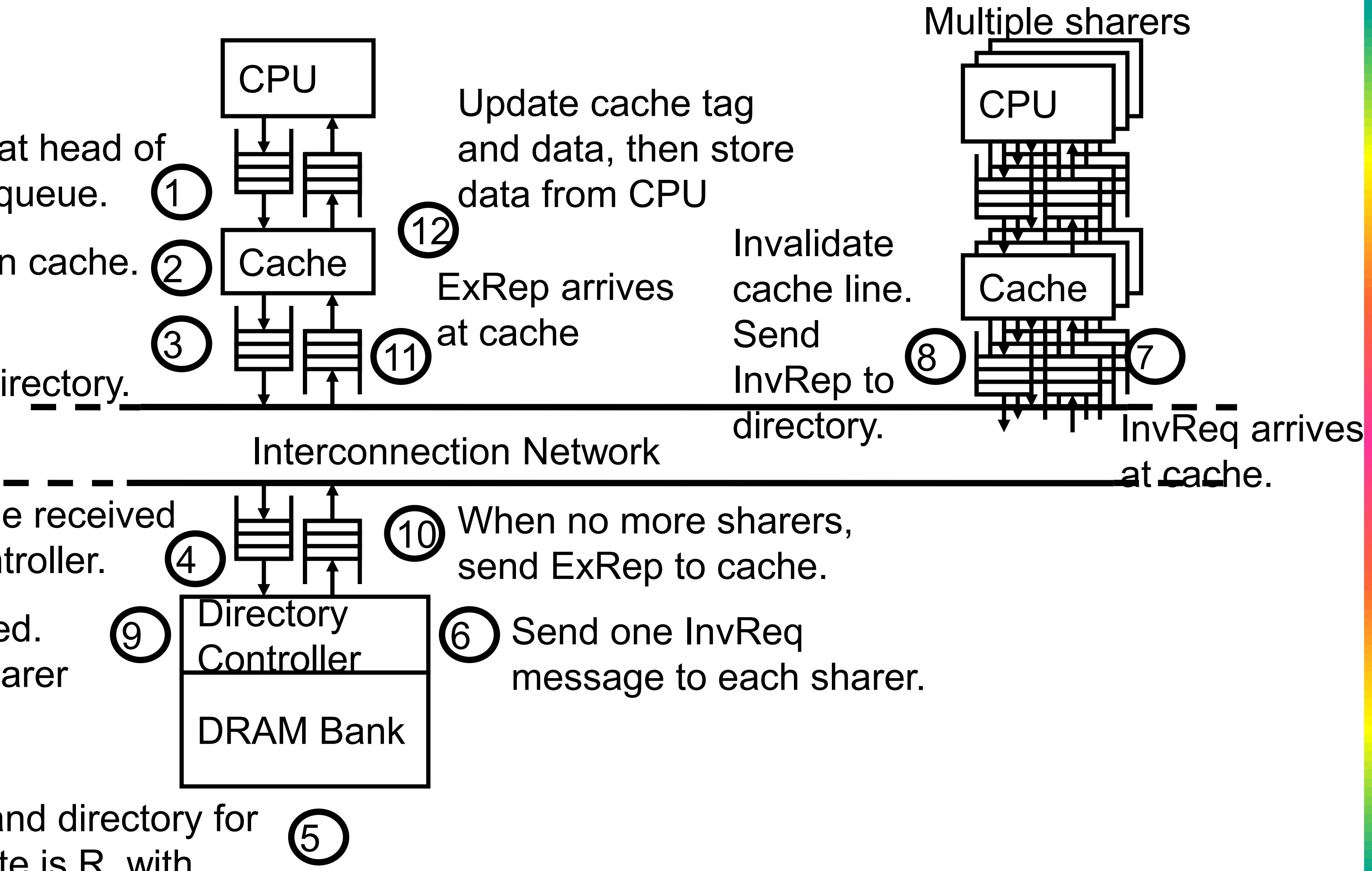
6 Update directory by setting bit for new processor sharer.

7 Send ShRep message with contents of cache line.

8 ShRep arrives at cache.

9 Update cache tag and data and return load data to CPU.

Write miss, to read shared line



1 Store request at head of CPU->Cache queue.

2 Store misses in cache.

3 Send ExReq message to directory.

4 ExReq message received at directory controller.

5 Access state and directory for line. Line's state is R, with some set of sharers.

6 Send one InvReq message to each sharer.

7 InvReq arrives at cache.

8 Invalidate cache line. Send InvRep to directory.

9 InvRep received. Clear down sharer bit.

10 When no more sharers, send ExRep to cache.

11 ExRep arrives at cache

12 Update cache tag and data, then store data from CPU

目录协议

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

目录协议（续）

- No bus and don’t want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - **Local node** where a request originates
  - **Home node** where the memory location of an address resides
  - **Remote node** has a copy of a cache block, whether exclusive or shared
- Example messages on next slide: P = processor number, A = address

目录协议消息

Message type	*Source*	*Destination*	*Msg Content*
Read miss	Local cache	Home directory	P, A
● *Processor P reads data at address A; make P a read sharer and arrange to send data back*			
Write miss	Local cache	Home directory	P, A
● *Processor P writes data at address A; make P the exclusive owner and arrange to send data back*			
Invalidate	Home directory	Remote caches	A
● *Invalidate a shared copy at address A.*			
Fetch	Home directory	Remote cache	A
● *Fetch the block at address A and send it to its home directory*			
Fetch/Invalidate	Home directory	Remote cache	A
● *Fetch the block at address A and send it to its home directory; invalidate the block in the cache*			
Data value reply	Home directory	Local cache	Data
● *Return a data value from the home memory (read miss response)*			
Data write-back	Remote cache	Home directory	A, Data
● *Write-back a data value for address A (invalidate response)*			

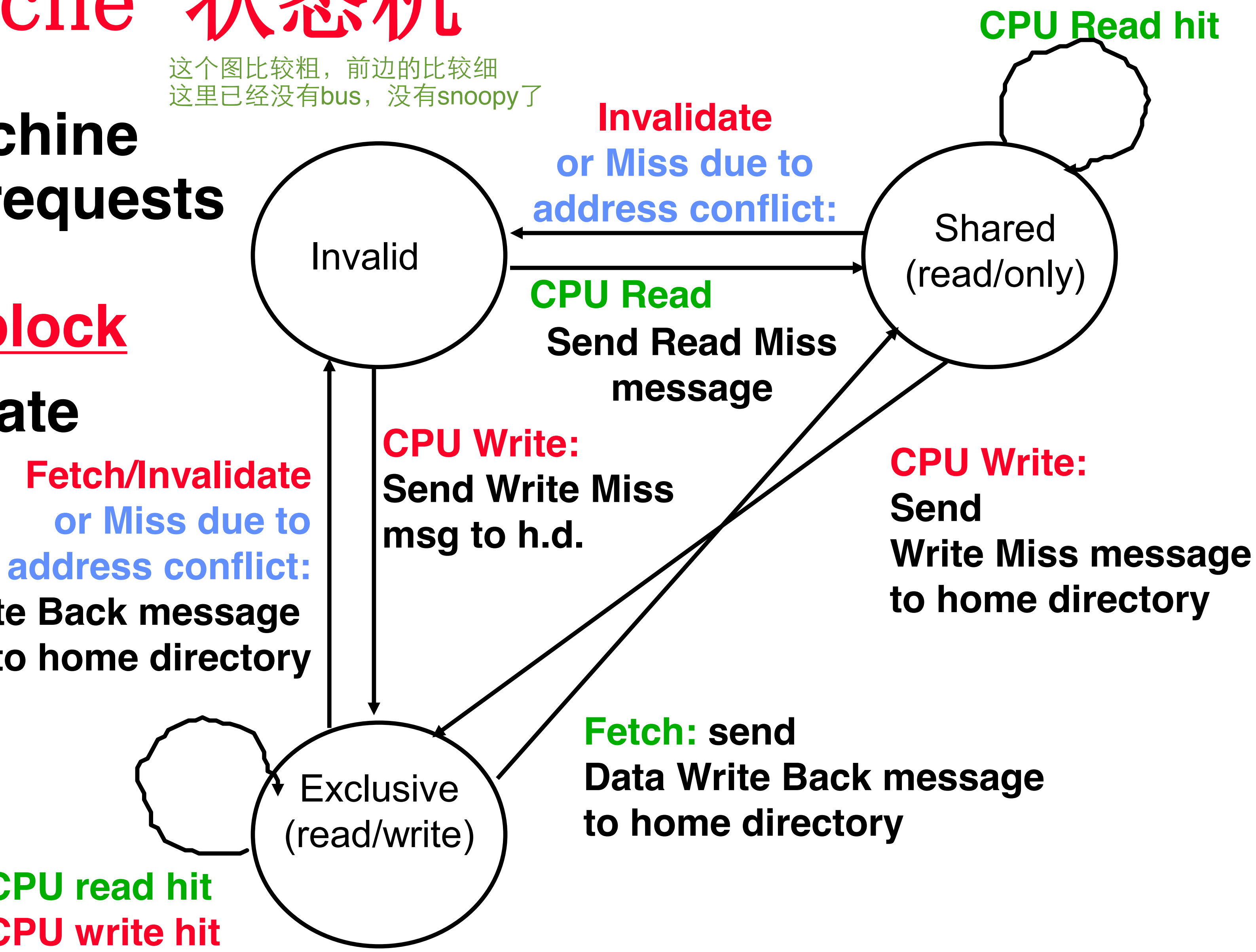
基于目录系统中一个独立Cache块的状态变换图

- States identical to snoopy case; transactions very similar.
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss msg to home directory.
- Write misses that were broadcast on the bus for snooping => explicit invalidate & data fetch requests.

CPU-Cache 状态机

这个图比较粗，前边的比较细
这里已经没有bus，没有snoopy了

- State machine for *CPU* requests for each memory block
- Invalid state if in memory

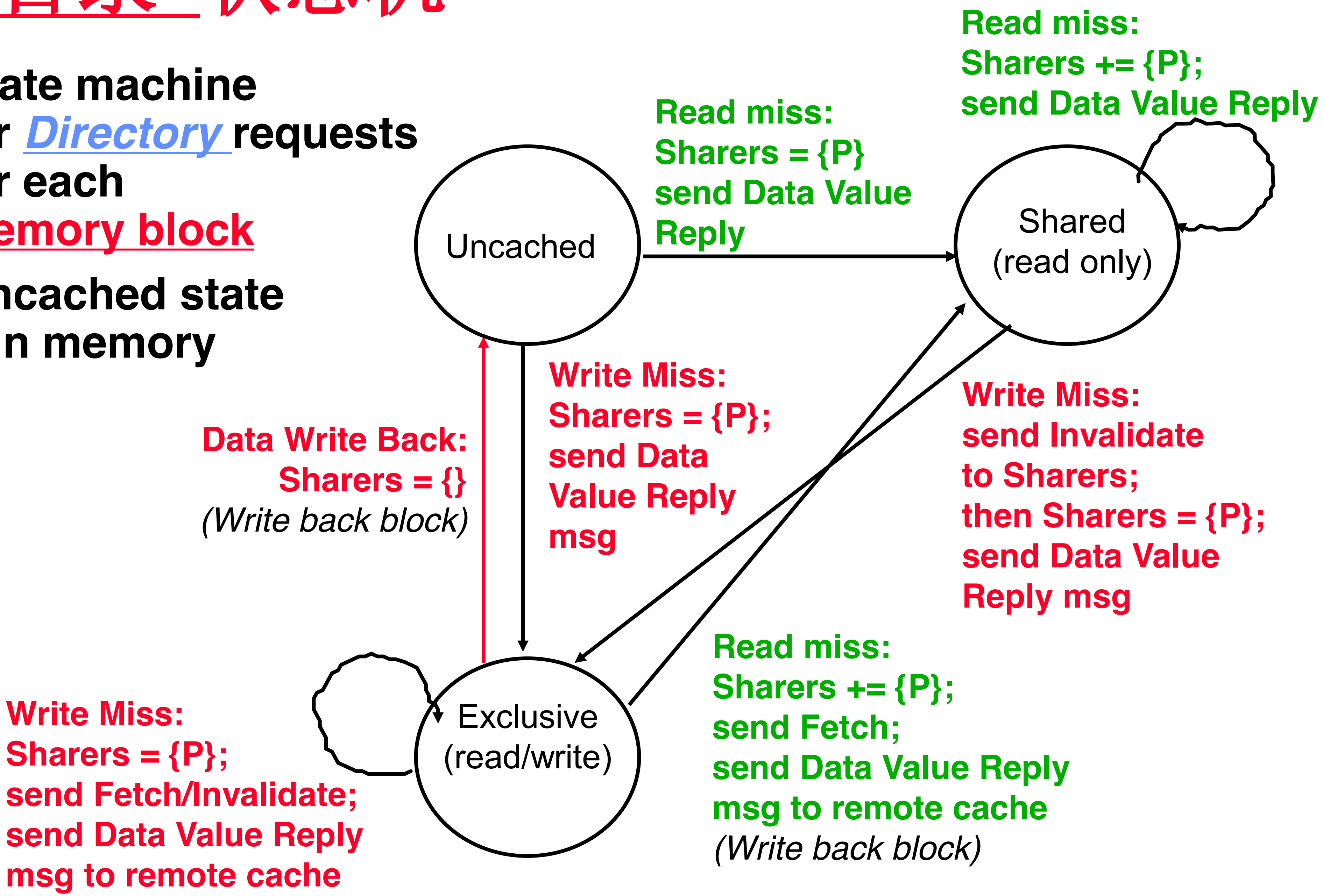


目录的状态变换图

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send msgs to satisfy requests
- Tracks all copies of memory block.
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message.

目录 状态机

- State machine for *Directory* requests for each **memory block**
- Uncached state if in memory



Uncached

Shared (read only)

Exclusive (read/write)

Read miss:
Sharers = {P}
send Data Value Reply

Read miss:
Sharers += {P};
send Data Value Reply

Write Miss:
Sharers = {P};
send Data Value Reply msg

Write Miss:
send Invalidate to Sharers;
then Sharers = {P};
send Data Value Reply msg

Data Write Back:
Sharers = {}
(Write back block)

Read miss:
Sharers += {P};
send Fetch;
send Data Value Reply msg to remote cache
(Write back block)

Write Miss:
Sharers = {P};
send Fetch/Invalidate;
send Data Value Reply msg to remote cache

目录协议示例

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request
- Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  - Read miss: requesting processor sent data from memory &requestor made only sharing node; state of block made Shared.
  - Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
- Block is Shared => the memory value is up-to-date:
  - Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  - Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

目录协议示例（续）

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
  - Read miss: owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor.
    Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
  - Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date
    (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
  - Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

示例

以前这里是讲bus，现在没有bus是interconnect

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2							Directory			Memo
step	Stat	Add	Valu	Stat	Add	Valu	Actio	Proc	Add	Valu	Add	Stat	{Procs	Value
P1 Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

示例（续二）

Processor 1　Processor 2　Interconnect　Directory　Memory

	P1			P2							Directory			Memo
step	Stat	Add	Valu	Stat	Add	Valu	Actio	Proc	Add	Valu	Add	Stat	{Proc	Value
P1 Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

示例（续三）

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			*P2*							*Directory*			*Memo*
step	*Stat*	*Add*	*Valu*	*Stat*	*Add*	*Valu*	*Actio*	*Proc*	*Add*	*Valu*	*Add*	*Stat*	*{Proc*	*Value*
P1 Write 10 to A1							*WrMs*	P1	A1		*A1*	*Ex*	*{P1}*	
	Excl.	*A1*	*10*				*DaRp*	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

示例（续四）

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2							Directory			Memo
step	Stat	Add	Valu	Stat	Add	Valu	Actio	Proc	Add	Valu	Add	Stat	{Proc	Value
P1 Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10				10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2	10
P2: Write 20 to A1														10
														10
P2: Write 40 to A2														10

Write Back

Write Back

A1 and A2 map to the same cache block

示例（续五）

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2							Directory			Memo
step	Stat	Add	Valu	Stat	Add	Valu	Actio	Proc	Add	Valu	Add	Stat	{Proc	Value
P1 Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10			A1	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2														10

A1 and A2 map to the same cache block

示例（续六）

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2							Directory			Memo
step	Stat	Add	Valu	Stat	Add	Valu	Actio	Proc	Add	Valu	Add	Stat	{Procs	Value
P1 Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10			A1	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2							WrMs	P2	A2		A2	Excl.	{P2}	0
							WrBk	P2	A1	20	A1	Jnca	{}	20
				Excl.	A2	40	DaRp	P2	A2	0	A2	Excl	{P2}	0

A1 and A2 map to the same cache block

至此，针对两种不同的多处理器结构，我们把它们的cache coherence做好了。
目前做到的是针对的是单一地址，不同处理器看到的存储器的地址是相同的

实现一个目录

- We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of bufffers in network
- Optimizations:
  - read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor

学了存储层次和多处理器之后，我们知道，读和写不是那么简单，并不是要多快有多快，并不是瞬间就读好或写好了，比如numa你都不知道读写要多长时间，你怎么能保证说一个处理器写完别的读就是它写的值呢？我们当然不能让处理器之间互相等，而是要让各处理器自己是独立的。以前我们假设程序就是串行的而且是在单一处理器、单一存储器上执行的，于是我的单一程序就有一个很好辨识的program order，一个程序序列，程序的执行序列和program order本身的顺序就是严格一致的。多个处理器上来，哪个先访问

另一多处理器论题：存储器一致性模型

consistency：你给我什么样的编程模型，你的底层一定要保证这个模型，以使得我的并行程序能做对。（两个不想干的程序之间我不关心）

- What is consistency? **When** must a processor see the new value before another processor update it? e.g., seems that

```
P1:   A = 0;                 P2:     B = 0;
       .....                          .....
      A = 1;                         B = 1;
L1:   if (B == 0) ...        L2:     if (A == 0) ...
```

- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- Memory consistency models: what are the rules for such cases?
- **Sequential consistency**: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
  - SC: delay all memory accesses until all invalidates done

存储器一致性模型

- Schemes faster execution to sequential consistency
- Not really an issue for most programs; they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations

```
write (x)
...
release (s) {unlock}
...
acquire (s) {lock}
...
read(x)
```

- Only those programs willing to be nondeterministic are not synchronized: “data race”: outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW
  to different addresses

Synchronization

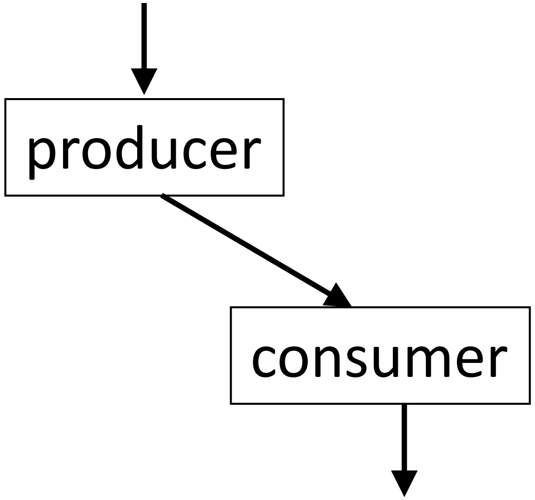
The need for synchronization arises whenever there are concurrent processes in a system

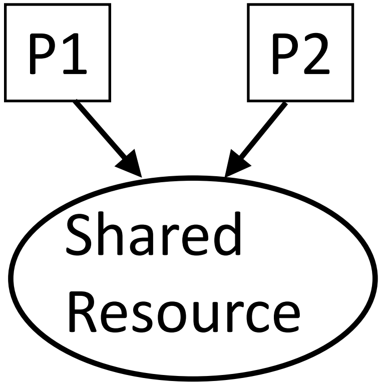
(even in a uniprocessor system)

Two classes of synchronization:

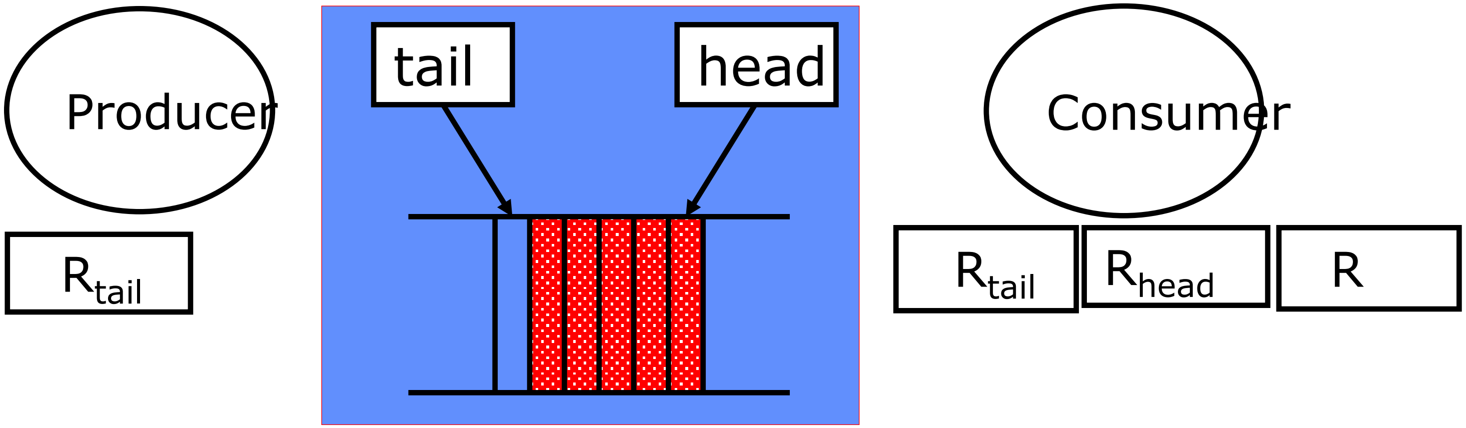
Producer-Consumer: A consumer process must wait until the producer process has produced data

Mutual Exclusion: Ensure that only one process uses a resource at a given time





A Producer-Consumer Example



Producer posting Item x:

```
        Load $R_{tail}$, (tail)
        Store ($R_{tail}$), x
        $R_{tail}$=$R_{tail}$+1
        Store (tail), $R_{tail}$
```

Consumer:

```
        Load $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$=$R_{head}$+1
        Store (head), $R_{head}$
        process(R)
```

乱序执行的时候，加锁这种事情怎么保证呢?

The program is written assuming instructions are executed in order.

Problems?

A Producer-Consumer Example

continued

Producer posting Item x:

```
        Load $R_{tail}$, (tail)
1       Store ($R_{tail}$), x
        $R_{tail}$=$R_{tail}$+1
2       Store (tail), $R_{tail}$
```

Consumer:

```
        Load $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)                3
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)                  4
        $R_{head}$=$R_{head}$+1
        Store (head), $R_{head}$
        process(R)
```

Can the tail pointer get updated before the item x is stored?

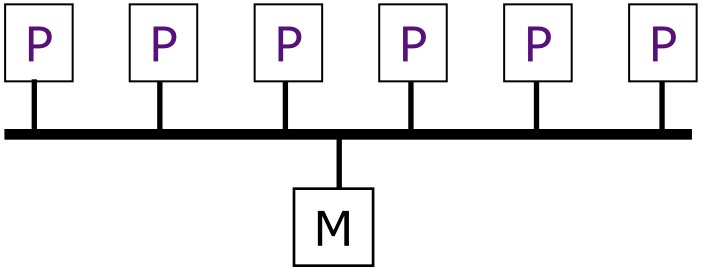
Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:

2, 3, 4, 1
4, 1, 2, 3

Sequential Consistency

A Memory Model



“ A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

Leslie Lamport

串行一致性：如果你把一个程序分开了，分到两个处理器上跑，不管你怎么在两个处理器上跑，它们都的结果都一样，并且和把这两个程序放在同一个处理器上跑结果是一样的。保证所有的load、stoe顺序和串行顺序一样即可？？ 但不能这样做，因为代价太大

Sequential Consistency =

arbitrary *order-preserving interleaving*

of memory references of sequential programs

Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:

Store (X), 1 *(X = 1)*
Store (Y), 11 *(Y = 11)*

T2:

Load R_1, (Y)
Store (Y'), R_1 *(Y'= Y)*
Load R_2, (X)
Store (X'), R_2 *(X'= X)*

what are the legitimate answers for X' and Y' ?

(X',Y') ε {(1,11), (0,10), (1,10), ~~(0,11)}~~ ?

If y is 11 then x cannot be 0

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (   ) →

What are these in our example ?

以前说能不能调整顺序，只要考虑传统的几个相关
现在还要加上sequential consistency
这样的话以前讲的多核，superscalar的技术就不能乱用了
cache coherent问题，以致引申到memory consistency问题，不能像以前superscalar一样去调整顺序了

T1:

Store (X), 1 *(X = 1)*
Store (Y), 11 *(Y = 11)*

T2:

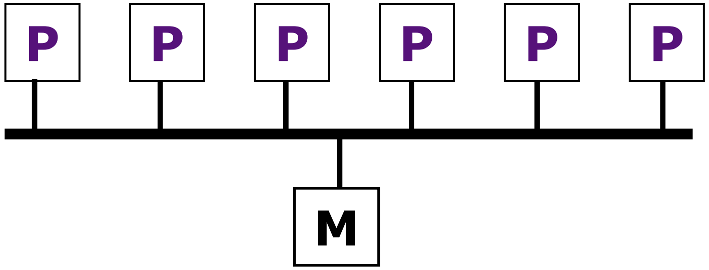
Load R_1, (Y)
Store (Y'), R_1 *(Y'= Y)*
Load R_2, (X)
Store (X'), R_2 *(X'= X)*

→ additional SC requirements

Does (can) a system with caches or out-of-order execution capability provide a *sequentially consistent* view of the memory ?

more on this later

Issues in Implementing Sequential Consistency



Implementation of SC is complicated by two issues

- *Out-of-order execution capability*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a ≠ b
Store(a); Load(b)	*yes if* a ≠ b
Store(a); Store(b)	*yes if* a ≠ b

- *Caches*

  Caches can prevent the effect of a store from being seen by other processors

No common commercial architecture has a sequentially consistent memory model!

Implementing SC in hardware

- Only a few commercial systems implemented SC
  - Neither x86 nor ARM are SC
- Requires either severe performance penalty
  - Wait for stores to complete before issuing new store
- Or, complex hardware
  - Speculatively issue loads but squash if memory inconsistency with later-issued store discovered (MIPS R10K)

Software reorders too!

```
//Producer code
*datap = x/y;
*flagp = 1;
```

```
//Consumer code
while (!*flagp)
       ;
d = *datap;
```

- Compiler can reorder/remove memory operations unless made aware of memory model
  - Instruction scheduling, move loads before stores if to different address
  - Register allocation, cache load value in register, don’t check memory
- Prohibiting these optimizations would result in very poor performance

Relaxed Memory Models

x86和arm都不支持sequential consistency，都只支持自己定义的一些简化

- Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies were needed
- Which dependencies are dropped depends on the particular memory model
  - IBM370, TSO, PSO, WO, PC, Alpha, RMO, …
- How to introduce needed dependencies varies by system
  - Explicit FENCE instructions (sometimes called sync or memory barrier instructions)
  - Implicit effects of atomic memory instructions

Memory Fences

Instructions to sequentialize memory accesses

Processors with *relaxed or weak memory models* (i.e., permit Loads and Stores to different addresses to be reordered) need to provide *memory fence* instructions to force the serialization of memory accesses

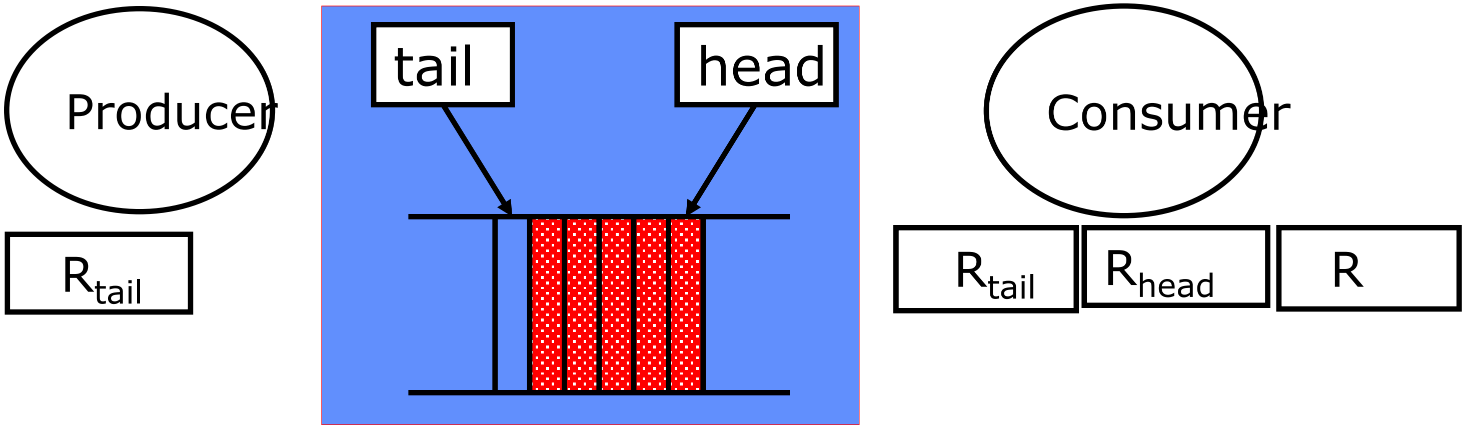
Examples of processors with relaxed memory models:

Sparc V8 (TSO,PSO): Membar
Sparc V9 (RMO):
Membar #LoadLoad, Membar #LoadStore
Membar #StoreLoad, Membar #StoreStore

PowerPC (WO): Sync, EIEIO
ARM: DMB (Data Memory Barrier)
X86/64: mfence (Global Memory Barrier)

Memory fences are expensive operations, however, one pays the cost of serialization only when it is required

Using Memory Fences



Producer posting Item x:

```
        Load $R_{tail}$, (tail)
        Store ($R_{tail}$), x
        $Membar_{SS}$
        $R_{tail}$=$R_{tail}$+1
        Store (tail), $R_{tail}$
```

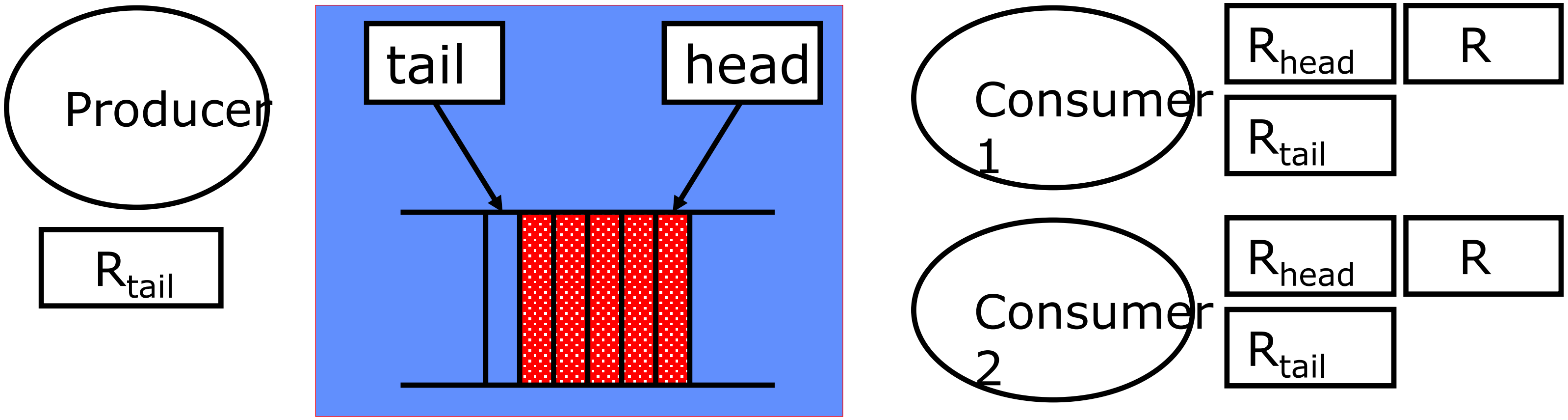
ensures that tail ptr is not updated before x has been stored

Consumer:

```
        Load $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)
        if $R_{head}$==$R_{tail}$ goto spin
        $Membar_{LL}$
        Load R, ($R_{head}$)
        $R_{head}$=$R_{head}$+1
        Store (head), $R_{head}$
        process(R)
```

ensures that R is not loaded before x has been stored

Multiple Consumer Example



Producer posting Item x:

```
Load $R_{tail}$, (tail)
Store ($R_{tail}$), x
$R_{tail}$=$R_{tail}$+1
Store (tail), $R_{tail}$
```

Consumer:

```
spin:  Load $R_{head}$, (head)
       Load $R_{tail}$, (tail)
       if $R_{head}$==$R_{tail}$ goto spin
       Load R, ($R_{head}$)
       $R_{head}$=$R_{head}$+1
       Store (head), $R_{head}$
       process(R)
```

Critical section:
Needs to be executed atomically by one consumer

What is wrong with this code?

Mutual Exclusion Using Load/Store

A protocol based on two shared variables c1 and c2.
Initially, both c1 and c2 are 0 *(not busy)*

Process 1

```
       …
       c1=1;
L:   if c2=1 then go to L
         < critical section>
       c1=0;
```

Process 2

```
       …
       c2=1;
L:   if c1=1 then go to L
         < critical section>
       c2=0;
```

What is wrong?   *Deadlock!*

Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

Process 1

```
     ...
L:  c1=1;
    if c2=1 then
        { c1=0; go to L}
      < critical section>
    c1=0
```

Process 2

```
     ...
L:  c2=1;
    if c1=1 then
        { c2=0; go to L}
      < critical section>
    c2=0
```

- Deadlock is not possible but with a low probability a *livelock* may occur.

- An unlucky process may never get to enter the critical section ⇒ *starvation*

A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and turn.
Initially, both c1 and c2 are 0 *(not busy)*

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

- turn = *i* ensures that only process *i* can wait
- variables c1 and c2 ensure *mutual exclusion*

Solution for n processes was given by Dijkstra and is quite tricky!

N-process Mutual Exclusion

Lamport’s Bakery Algorithm

Process i

Initially num[j] = 0, for all j

Entry Code

```
        choosing[i] = 1;
        num[i] = max(num[0], …, num[N-1]) + 1;
        choosing[i] = 0;

        for(j = 0; j < N; j++)  {
             while( choosing[j] );
             while( num[j] &&
                        ( ( num[j] < num[i] ) ||
                          ( num[j] == num[i] && j < i ) ) );
        }
```

Exit Code

```
        num[i] = 0;
```

Locks or Semaphores

E. W. Dijkstra, 1965

A *semaphore* is a non-negative integer, with the following operations:

P(s): *if s>0, decrement s by 1, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P's and V's must be executed atomically, i.e., without

- *interruptions* or
- *interleaved accesses to s* by other processors

```
Process i
    P(s)
        <critical section>
    V(s)
```

initial value of s determines the maximum no. of processes in the critical section

Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:

atomic read-modify-write instructions

Examples: *m is a memory location, R is a register*

```
Test&Set (m), R:
    $R \leftarrow M[m]$;
    if $R==0$ then
        $M[m] \leftarrow 1$;
```

```
Fetch&Add (m), $R_V$, R:
    $R \leftarrow M[m]$;
    $M[m] \leftarrow R + R_V$;
```

```
Swap (m), R:
    $R_t \leftarrow M[m]$;
    $M[m] \leftarrow R$;
    $R \leftarrow R_t$;
```

活锁：没有死锁，但是俩进程都不停地测试之类的，不断运行但不干正事儿

在指令系统定义的时候，就要考虑编译和操作系统怎么做才能实现meomry consistency，然后你才能去选择你合适的model，还要考虑你硬件上的实现难度。

intel、arm等支持多核不一样，但有些共同遵守的东西

load-reserve & store-conditional 这个指令对支持好之后，那些test and set 之类的指令都可以用这个指令对来实现

Multiple Consumers Example

using the Test&Set Instruction

```
P:       Test&Set (mutex),$R_{temp}$
         if ($R_{temp}$!=0) goto P
         Load $R_{head}$, (head)
spin:    Load $R_{tail}$, (tail)
         if $R_{head}$==$R_{tail}$ goto spin
         Load R, ($R_{head}$)
         $R_{head}$=$R_{head}$+1
         Store (head), $R_{head}$
V:       Store (mutex),0
         process(R)
```

Critical Section

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P's and V's

What if the process stops or is swapped out while in the critical section?

Nonblocking Synchronization

```
Compare&Swap(m), $R_t$, $R_s$:
  if ($R_t$==M[m])
      then  M[m]=$R_s$;
            $R_s$=$R_t$ ;
            status ← success;
      else  status ← fail;
```

status is an *implicit argument*

```
try:    Load $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{newhead}$ = $R_{head}$+1
        Compare&Swap(head), $R_{head}$, $R_{newhead}$
        if (status==fail) goto try
        process(R)
```

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs’
            reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-reserve $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$ = $R_{head}$ + 1
        Store-conditional (head), $R_{head}$
        if (status==fail) goto try
        process(R)
```

Load-linked & Store-conditional

- Special register(s) to hold reservation flag and address, and the outcome of store-conditional
- *load linked* or *load locked* and *store conditional*.
- These instructions are used in sequence:
  - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails.
  - If the processor does a context switch between the two instructions, then the store conditional also fails.
  - The store conditional is defined to return 1 if it was successful and a 0 otherwise. Since the load linked returns the initial value and the store conditional returns 1 only if it succeeds,

```
try:    MOV R3,R4 ;mov exchange value
        LL R2,0(R1);load linked
        SC R3,0(R1);store conditional
        BEQZR3,try ;branch store fails
        MOV R4,R2 ;put load value in R4
```

an atomic exchange on the memory location specified by the contents of R1

Atomic fetch-and-increment based on LLSC

```
try:    LL R2,0(R1) ;load linked
        DADDUIR3,R2,#1 ;increment
        SC R3,0(R1) ;store conditional
        BEQZ R3,try ;branch store fails
```

- These instructions are typically implemented by keeping track of the address specified in the LL instruction in a register, often called the *link register.*
- If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (for example, by another SC), the link register is cleared.
- The SC instruction simply checks that its address matches that in the link register. If so, the SC succeeds; otherwise, it fails.