Computer Architectures

Multiprocessors & Consistency

Giorgio Richelli

Flynn's Taxonomy

- Flynn classified by data and control streams in 1966

Single Instruction Single Data (SISD) (Uniprocessor)	Single Instruction Multiple Data SIMD (CM-2, Vector, SSE/VMX)
Multiple Instruction Single Data (MISD) (????)	Multiple Instruction Multiple Data MIMD (Clusters, SMP servers)

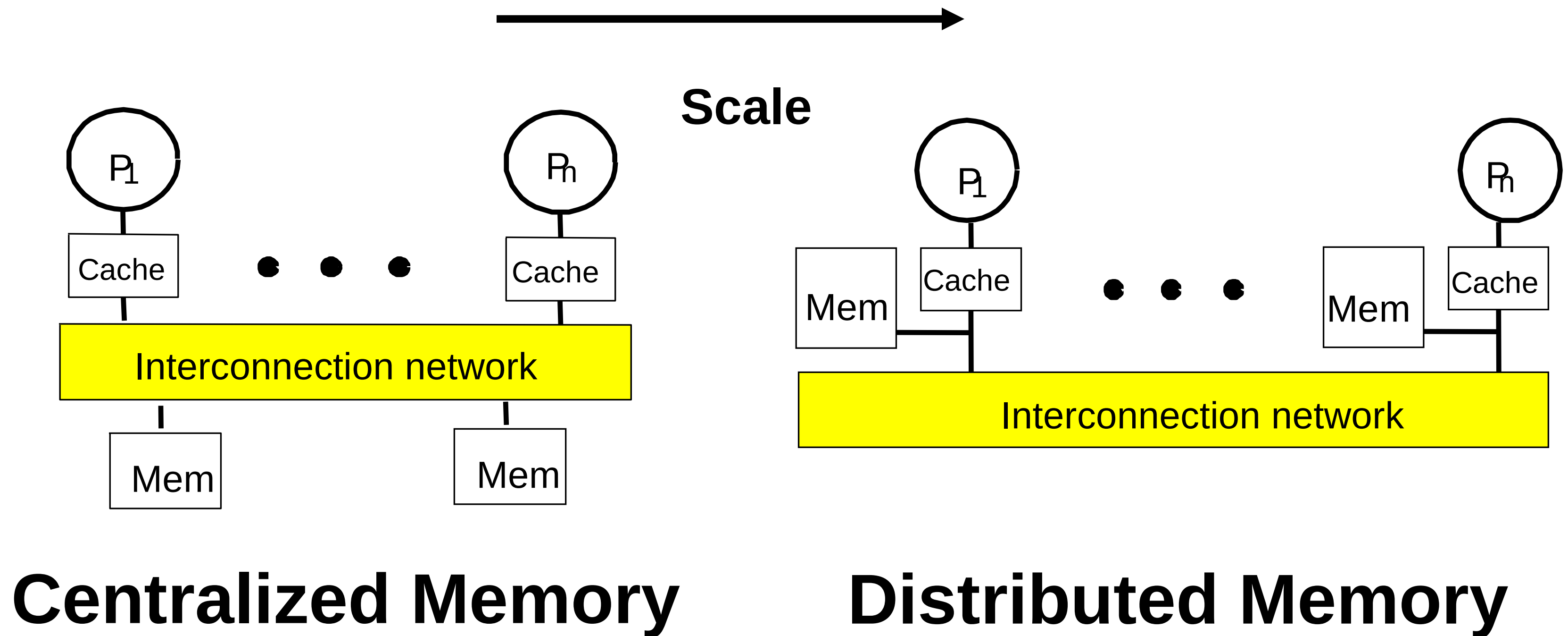
- SIMD $\Rightarrow$ Data Level Parallelism
- MIMD $\Rightarrow$ Thread Level Parallelism
- MIMD popular because
  - Flexible: N pgms and/or 1 multithreaded pgm
  - Cost-effective: same CPU as for desktops

Back to Basics

- "A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."
- Parallel Architecture = Computer Architecture + Communication Architecture
- 2 classes of multiprocessors WRT memory:

1. Centralized Memory Multiprocessor
   - Few dozen chips and ~ 100 cores
   - Small enough to share single, centralized memory
2. Physically Distributed-Memory Multiprocessor
   - Larger number of chips and cores than above.
   - BW demands $\Rightarrow$ Memory distributed among processors

Centralized vs. Distributed Memory



Centralized Memory **Distributed Memory**

Centralized-Memory Multiprocessors

- Also called symmetric multiprocessors (SMPs) because single main memory has symmetric relationship to all processors
- Large caches $\Rightarrow$ single memory can satisfy memory demands of small number of processors
- Can scale to few dozen processors by using a switch and many memory banks
- Further scaling technically conceivable but becomes less attractive as number of processors sharing centralized memory increases

Distributed-Memory Multiprocessors

- Pro: Cost-effective way to scale memory bandwidth
  - If most accesses are to local memory
- Pro: Reduces latency of local memory accesses
- Con: Communicating data between processors more complex
- Con: Could have to change software

Two Models for Communication and Memory Architecture

1. Communication occurs by explicitly passing messages among the processors:
   message-passing multiprocessors
2. Communication occurs through shared address space (via loads and stores):
   shared memory multiprocessors either
   - UMA (Uniform Memory Access time) for shared address, centralized memory MP
   - NUMA (Non Uniform Memory Access time multiprocessor) for shared address, distributed memory MP

- In past, confusion whether “sharing” means sharing physical memory (UMA) or sharing address space (NUMA)

Challenges of Parallel Processing

- First challenge is percentage of program that is inherently sequential
- Suppose we need 80X speedup from 100 processors. What fraction of original program can be sequential?
  a. 10%
  b. 5%
  c. 1%
  d. <1%

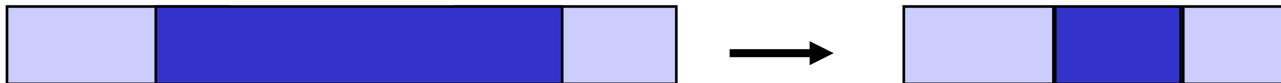
Amdahl's Law

$$\text{ExTime}_{new} = \text{ExTime}_{old} \times \left[\left(1 - \text{Fraction}_{enhanced}\right) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{ExTime}_{old}}{\text{ExTime}_{new}} = \frac{1}{\left(1 - \text{Fraction}_{enhanced}\right) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

Best you could ever hope to do:

$$\text{Speedup}_{maximum} = \frac{1}{\left(1 - \text{Fraction}_{enhanced}\right)}$$



Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{t_{serial}}{t_{parallel}} =$$

$$\frac{t_{serial}}{t_{serial} * \left(\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}} \right)}$$

$$80 = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 * \left[\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100} \right] = 1$$

$$79 = 80 * \text{Fraction}_{\text{parallel}} - 0.8 * \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

Challenges of Parallel Processing

- Second challenge is long latency to remote memory
- Suppose 32 CPU MP, 2GHz, 200 ns remote memory, all local accesses hit memory hierarchy and base CPI is 0.5. (Remote access = 200/0.5 = 400 clock cycles.)
- What is performance impact if 0.2% instructions involve remote access?
  a. < 1.5X
  b. 2.0X
  c. > 2.5X

CPI Equation

- CPI_{new} = CPI_{base}+
  RemoteAccessRate * RemoteRequestCost

- CPI_{new} = 0.5 + 0.2% * 400 = 0.5 + 0.8 = 1.3

- No communication is

CPI_{base}/CPI_{new}= 1.3/0.5 = 2.6 times faster

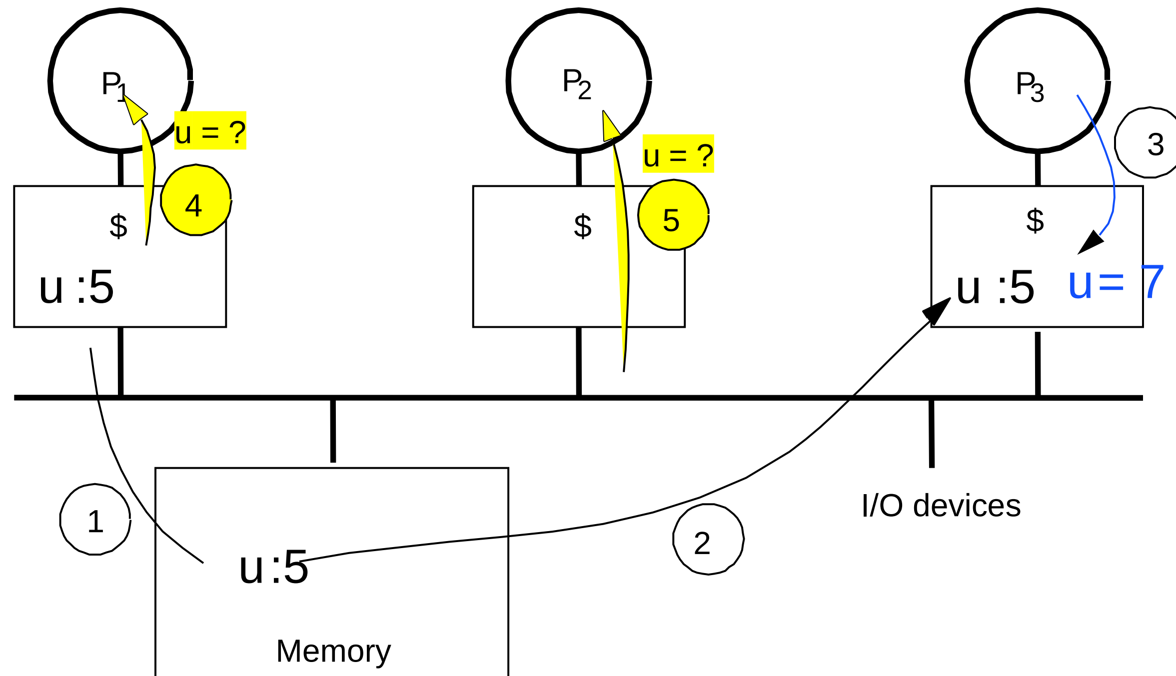
Symmetric Shared-Memory Architectures

- Caches
  - helps to alleviate memory latency
  - can be:
    - Dedicated to a single CPU
    - Shared among CPUs
  - contains
    - Private data used by a single processor
    - Shared data used by multiple processors
- Caching shared data

  ⇒ Reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth

  ⇒ Introduces cache coherence problem

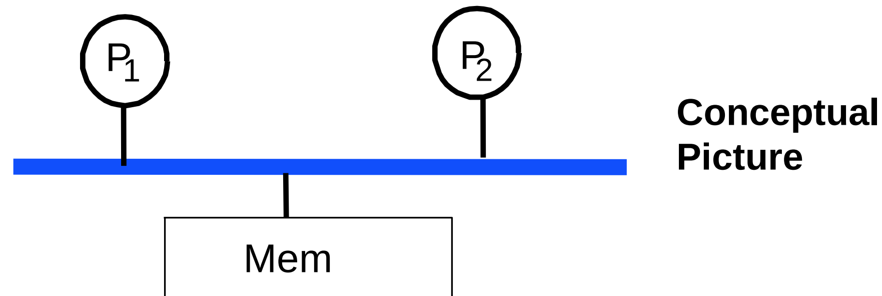
Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write-back caches, value written back to memory depends on which cache flushes or writes back value first
  - Processes accessing main memory may see stale value
  - Unacceptable for programming

Example (1)

- *Coherence* pertains only to *single location*
- Program behaviour could be different from what expected



P_1	/*Assume A=flag=0*/	P_2

```
A = 1;                        while (flag == 0);
flag = 1;                     print A;
```

Example (2)

<u>P1</u>

Flag1 = 1

if (Flag2 == 0)

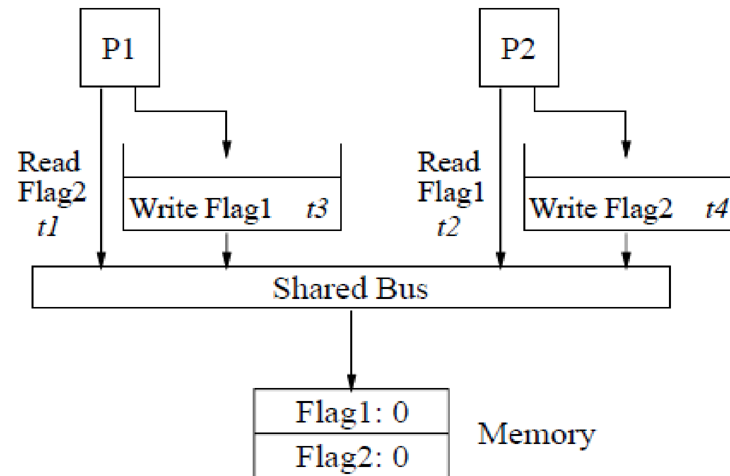
critical section

<u>P2</u>

Flag2 = 1

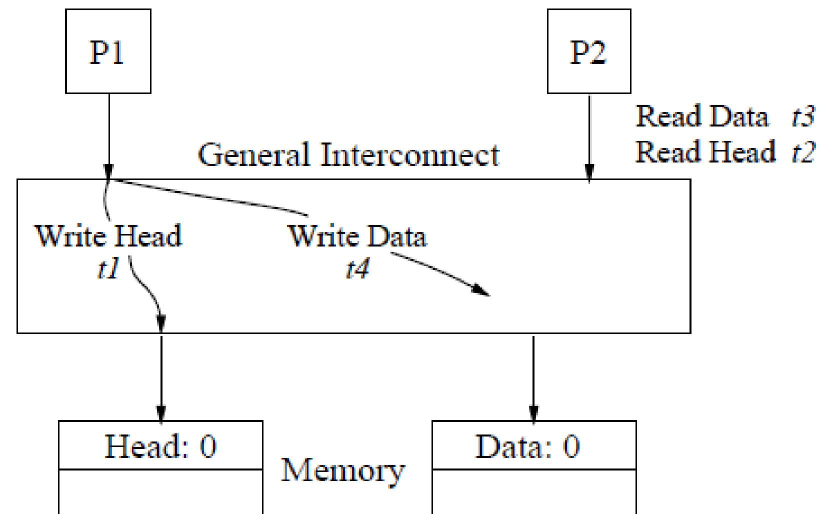
if (Flag1 == 0)

critical section



Example (3)

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data



Example (4)

P1

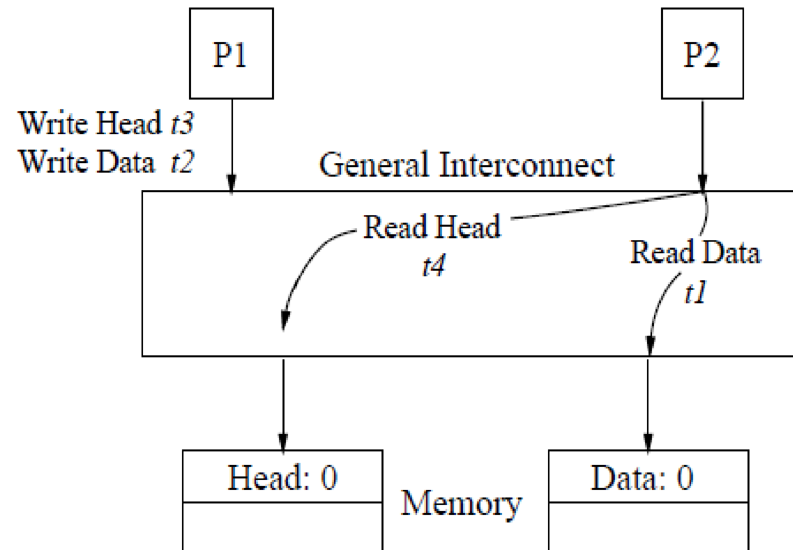
Data = 2000

Head = 1

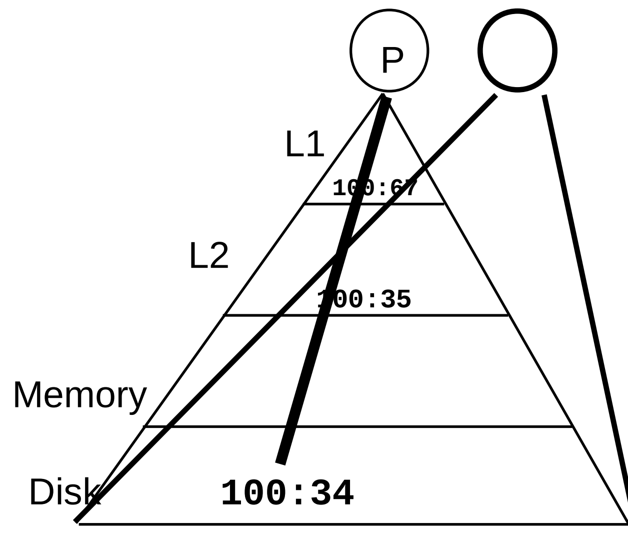
P2

while (Head == 0) {;}

... = Data



Intuitive Memory Model



- Reading an address should return the last value written to that address
  - Easy in uniprocessors, except for I/O

- Too vague and simplistic; 2 issues
  1. Coherence defines values returned by a read
  2. Consistency determines when a written value will be returned by a read
- Coherence defines behavior to *same* location
- Consistency defines behavior to *other* locations

Defining Coherent Memory System

1. Preserve Program Order: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. Coherent view of memory: Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
3. Write serialization: 2 writes to same location by any 2 processors are seen in the same order by all processors
   - If not, a processor could keep value 1 since saw as last write
   - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1

Write Consistency

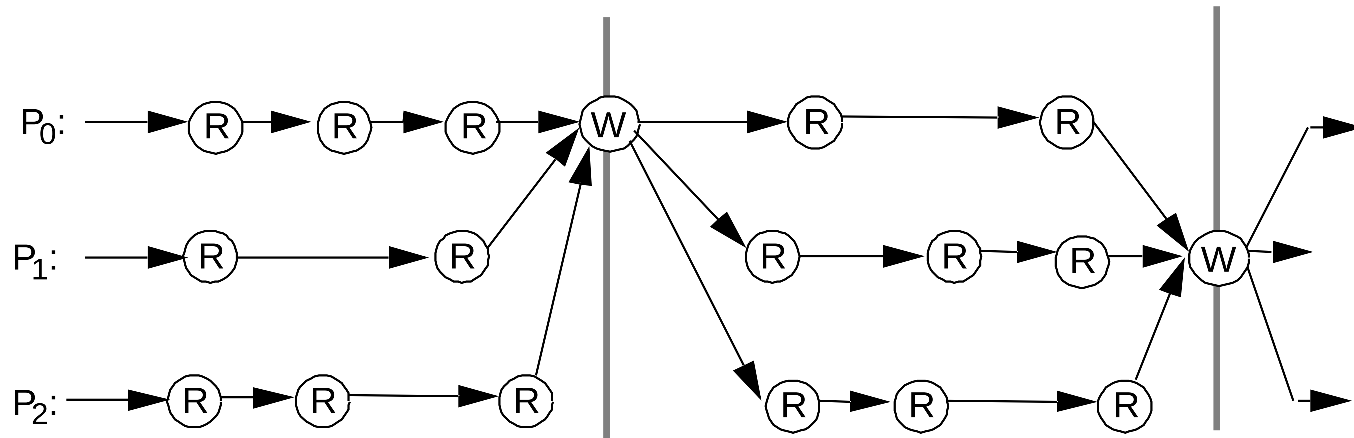
- Assume:

  1. A write does not complete (and allow the next write to occur) until all processors have seen effect of that write
  2. Processor does not change the order of any write with respect to any other memory access

  $\Rightarrow$ if a processor writes location A followed by location B, any processor that sees new value of B must also see new value of A

- These restrictions allow processor to reorder reads, but force it to finish writes in program order

Ordering



- Writes establish a partial order
- Doesn’t constrain ordering of reads, though
  a shared medium (bus) will order read misses too
  - Any order among reads between writes is fine,
    as long as in program order

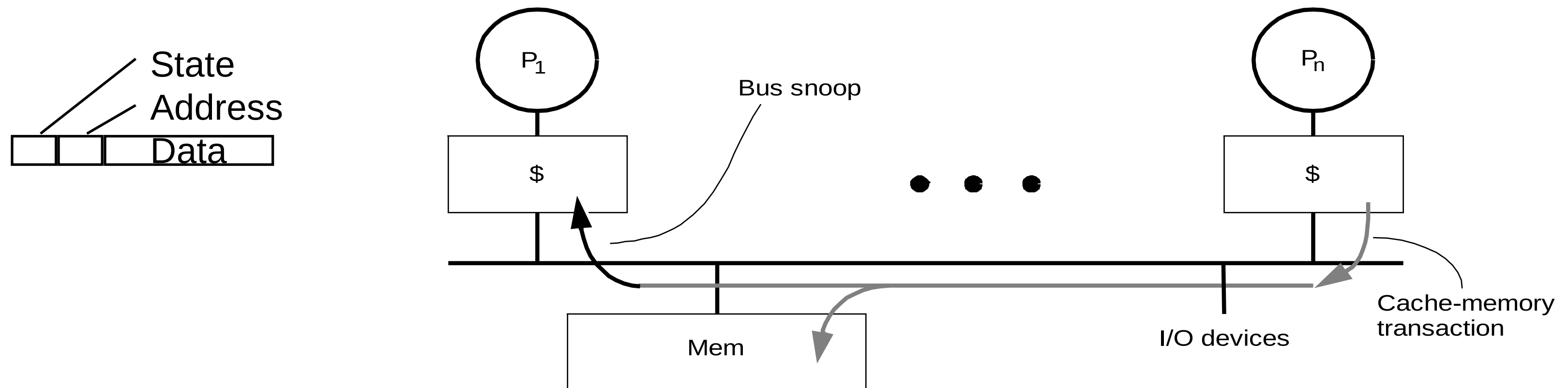
Basic Schemes for Enforcing Coherence

- Program on multiple processors will normally have copies of same data in several caches
  - Unlike I/O, where it's rare
- Rather than trying to avoid sharing in SW, SMPs use HW protocol to keep caches coherent
  - Migration and replication key to performance of shared data
- Migration - data can be moved to a local cache and used there in transparent fashion
  - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on shared memory
- Replication – for shared data being simultaneously read, since caches make copy of data in local cache
  - Reduces both latency of access and contention for read-

Classes of Cache Coherence Protocols

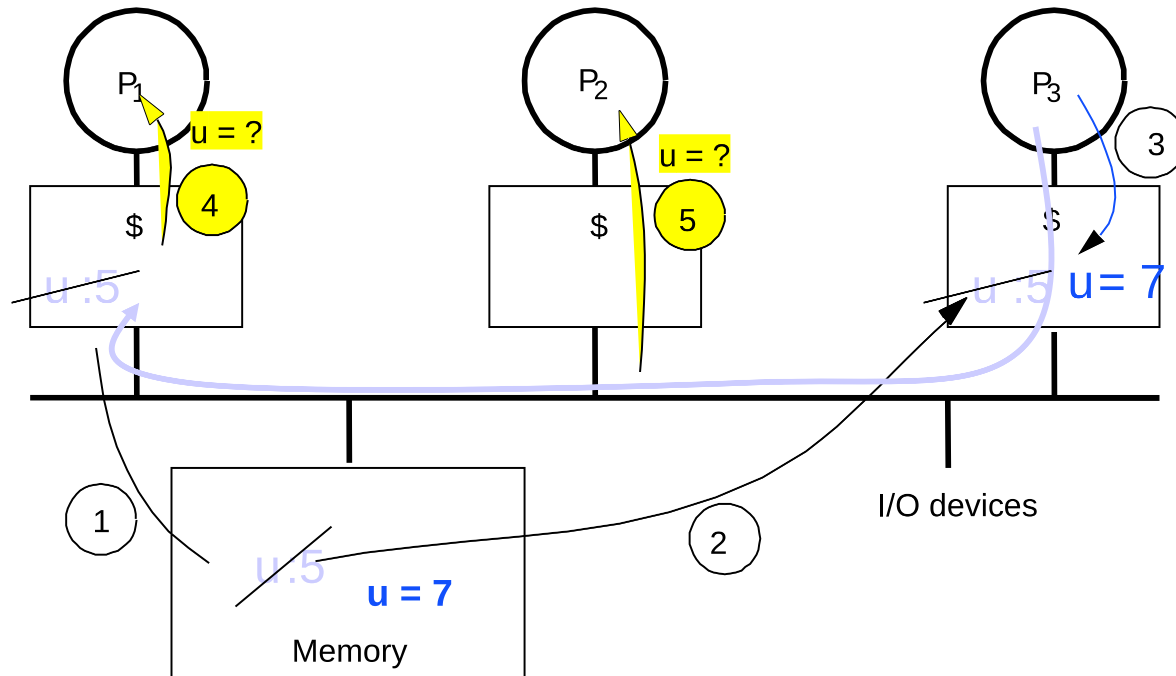
1. Directory-based — Sharing status of a block of physical memory is kept in just one location, the directory
2. Snooping — Every cache with copy of data also has copy of sharing status of block, but no centralized state is kept
   - All caches are accessible via some broadcast medium (a bus or switch)
   - All cache controllers monitor or snoop on the medium to determine whether or not they have copy of a block that is requested on bus or switch access

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or switch)
  - Relevant transaction if is for a block the cache contains
  - Take action to ensure coherence
    - Invalidate, update, or supply value
  - Depends on state of block and on protocol
- Either get exclusive access before write (via write invalidate), or update all copies on write

Example: Write-Thru Invalidate



- Must invalidate after step 3
- Write update uses more broadcast medium BW
  $\Rightarrow$ All recent CPUs use write invalidate

Locate Up-to-Date Copy of Data

- Write-through: get up-to-date copy from memory
  - Write through simpler if enough memory BW
- Write-back harder
  - Most recent copy can be in a cache
- Can use same snooping mechanism
  1. **Snoop every address placed on the bus**
  2. **If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access**
  - **Complexity from retrieving cache block from a processor cache, which can take longer than retrieving it from memory**
- Write-back needs lower memory bandwidth

  $\Rightarrow$ Support larger numbers of faster processors

  $\Rightarrow$ Most multiprocessors use write-back

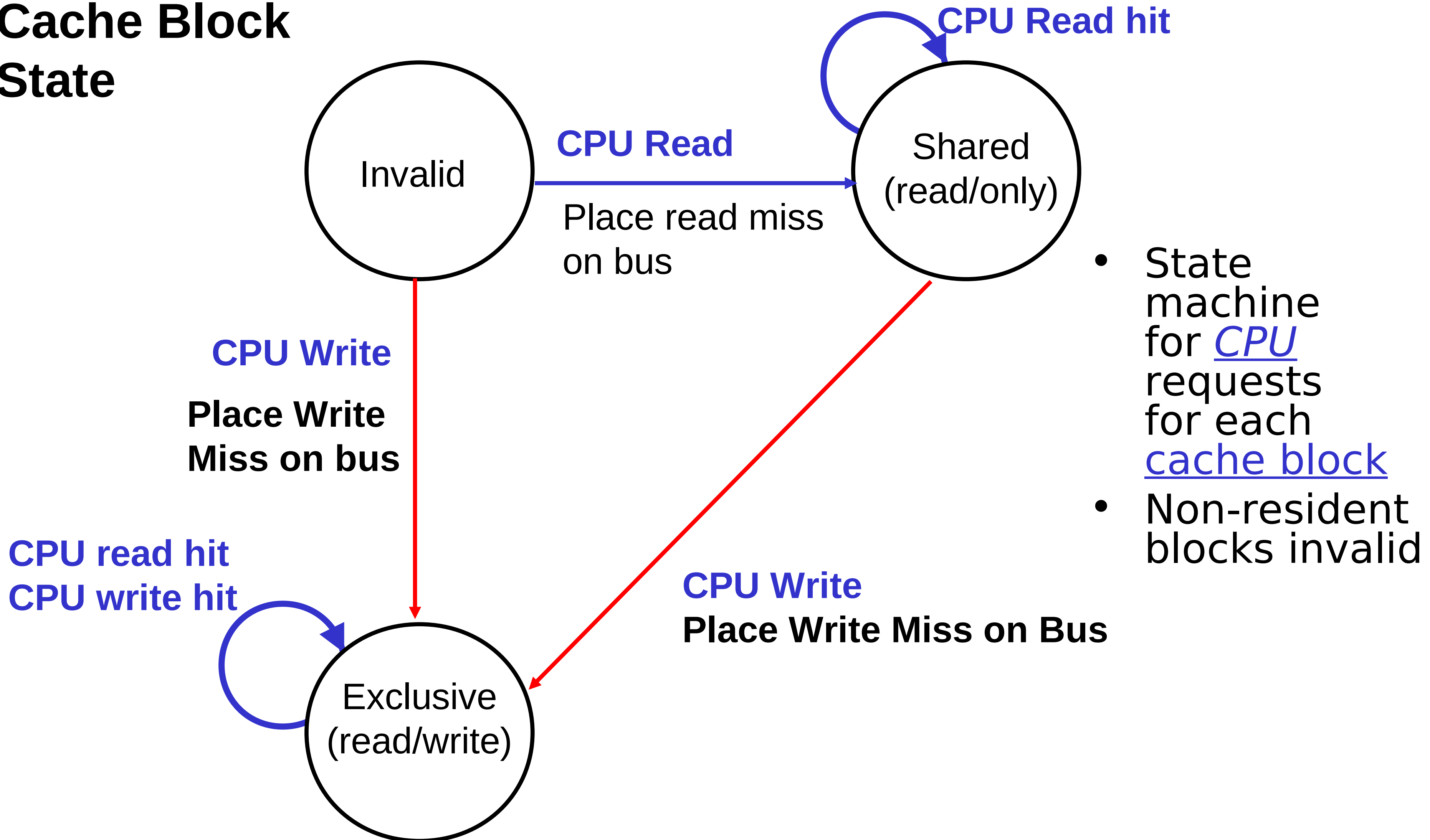
Example Protocol

- Snooping coherence protocol is usually implemented by incorporating finite-state controller in each node
- Logically, think of separate controller associated with each cache block
  - So snooping operations or cache requests for different blocks can proceed independently
- In reality, single controller allows multiple operations to distinct blocks to be interleaved
  - One operation may be initiated before another is completed even through only one cache or bus access allowed at a time

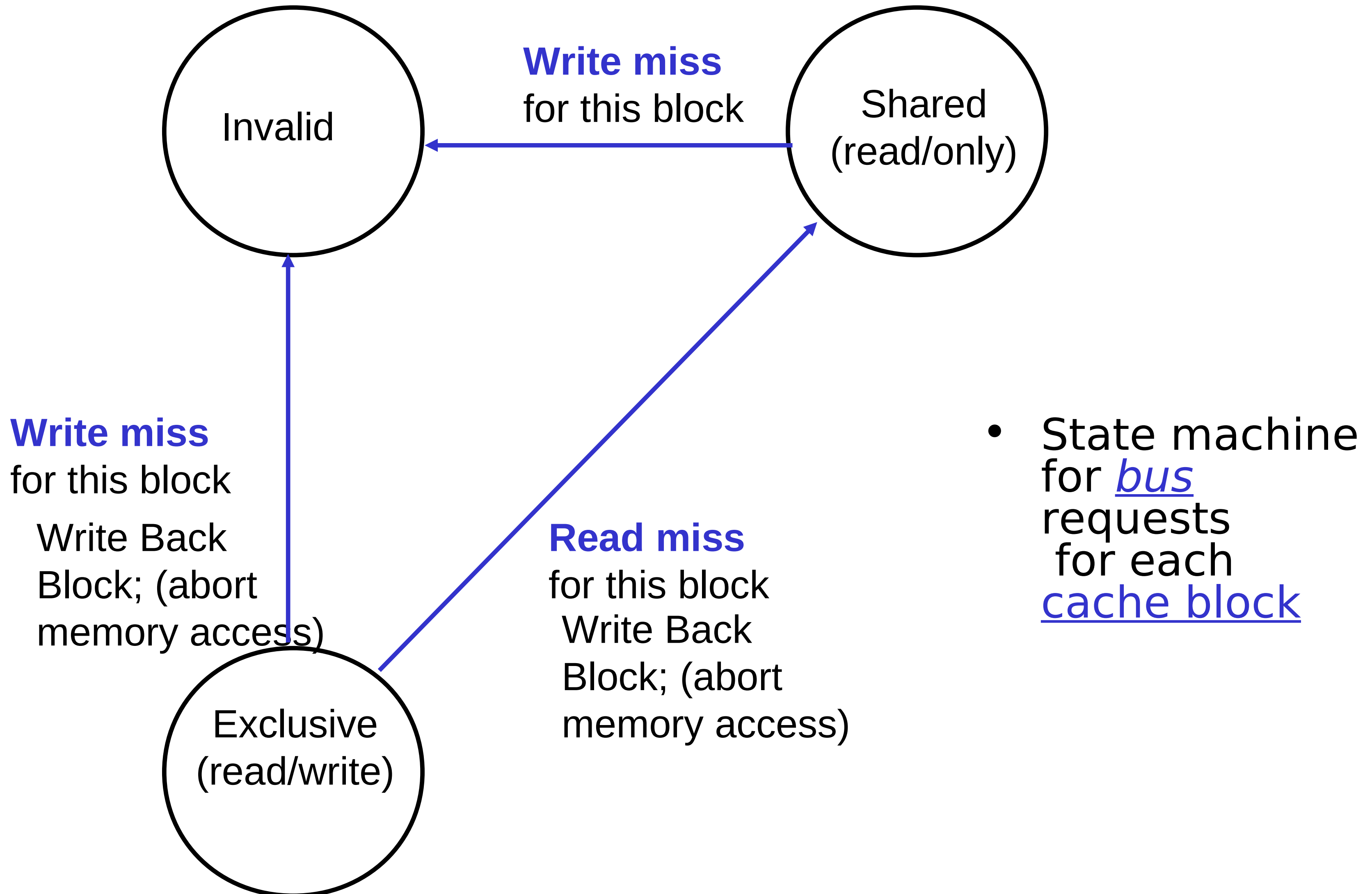
Example Write Back Snoopy Protocol

- Invalidation protocol, write-back cache
  - Snoops every address on bus
  - If has dirty copy of requested block, provides it in response to read request and aborts the memory access
- Each memory block is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR dirty in exactly one cache (Exclusive)
  - OR not in any caches
- Each cache block is in one state:
  - Shared : block can be read
  - OR Exclusive : cache has only copy, it’s writable and dirty
  - OR Invalid : block contains no data (in uniprocessor cache too)
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses

Write-Back State Machine - CPU

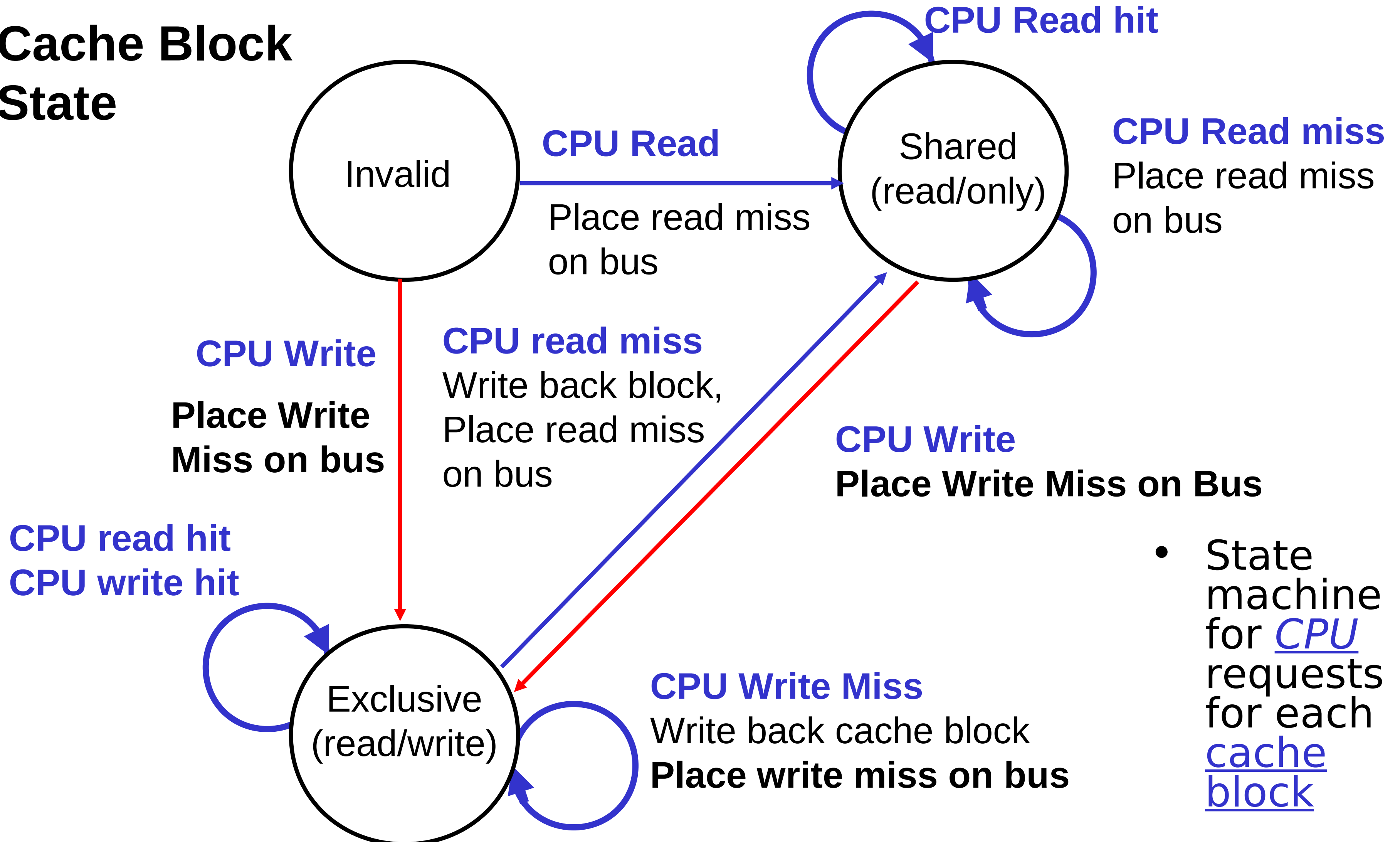


Write-Back State Machine- Bus request

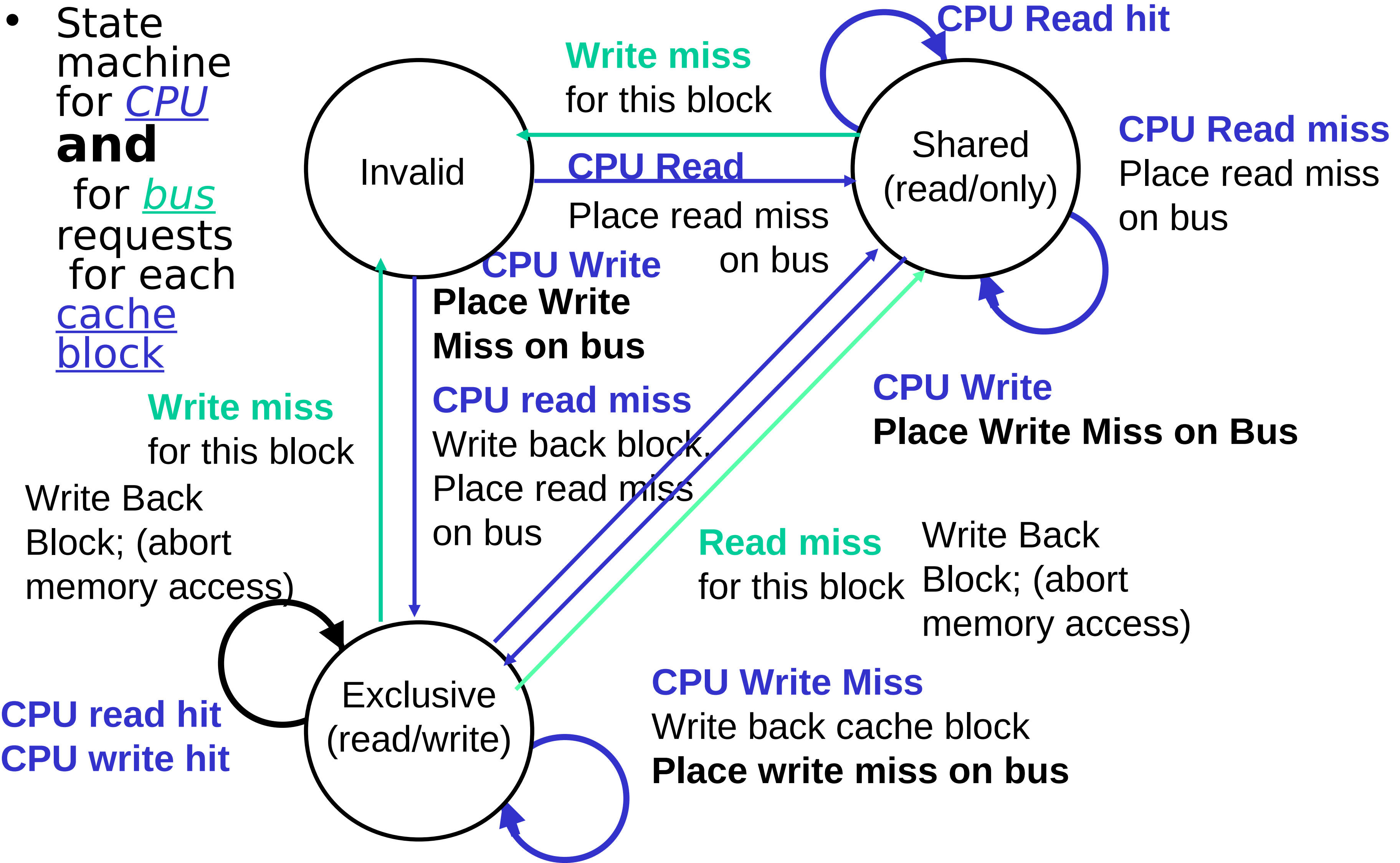


- State machine for *bus* requests for each cache block

Block Replacement



Write-back State Machine-III



Example

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

Example

	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 A1 and A2 map to same cache block

Example

	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 A1 and A2 map to same cache block

Example

	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	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	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	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	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	A1	*10*
				Shar.	A1	*10*	*RdDa*	P2	A1	10	A1	10
P2: Write 20 to A1	*Inv.*			*Excl.*	A1	*20*	*WrMs*	P2	A1		A1	10
P2: Write 40 to A2							*WrMs*	P2	A2		A1	10
				Excl.	*A2*	*40*	*WrBk*	P2	A1	20	A1	*20*

Assumes A1 and A2 map to same cache block,
but A1 != A2

MOESI

- **A protocol that encompasses all of the possible states commonly used in other protocols. Each cache line is in one of five states:**
  - **Modified**: A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.
  - **Owned**: A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent, correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state—all other processors must hold the data in the shared state.
  - **Exclusive**: A cache line in the exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.
  - **Shared**: A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. The copy in main memory is also the most recent, correct copy of the data, if no other processor holds it in owned state.
  - **Invalid**: A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another processor cache.