High Performance Computing

2. An Introduction to Parallel Architectures

Andrea Marongiu
(andrea.marongiu@unimore.it)
AA 2018-2019

Impact of Parallel Architectures

- From cell phones to supercomputers
- In regular CPUs as well as GPUs

1.4GHz Quad core Exynos (ARM)

Using multiple processor cores

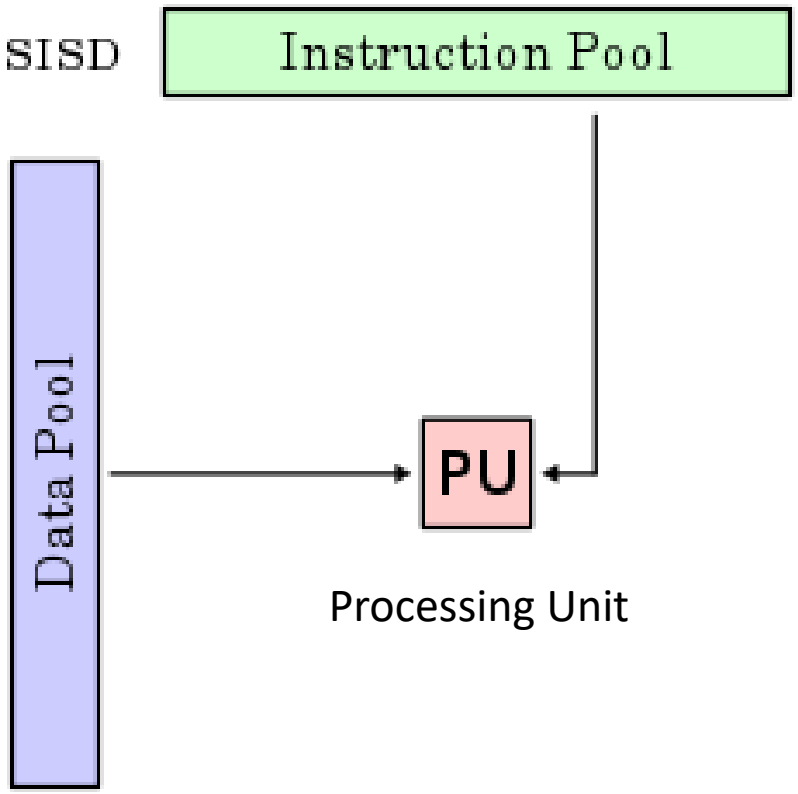
- Advantages:
  - Effective use of billion of transistors
    - Easier to reuse a basic unit many times
  - Potential for very easy scaling
    - Just keep adding cores for higher (peak) performance
  - Each individual processor can be less powerful
    - Which means it's cheaper to buy and run (less power)
- Disadvantages
  - Parallelization is not a limitless way to infinite performance!
  - Algorithms and computer hardware give limits on performance
  - One task – many processors
    - We need to think about how to share the task amongst them
    - We need to co-ordinate carefully
  - **We need a new way of writing our programs**

Taxonomy of parallel architectures

Flynn Taxonomy of parallel computers

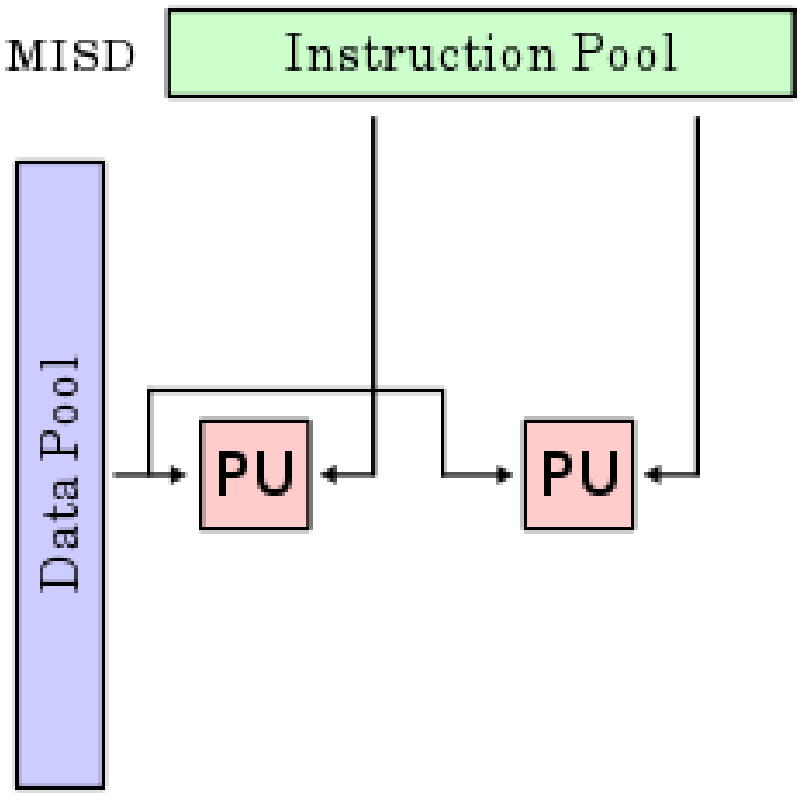
		Data streams	
		Single	Parallel
Instruction Streams	Single	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

Alternative Kinds of Parallelism: Single Instruction/Single Data Stream



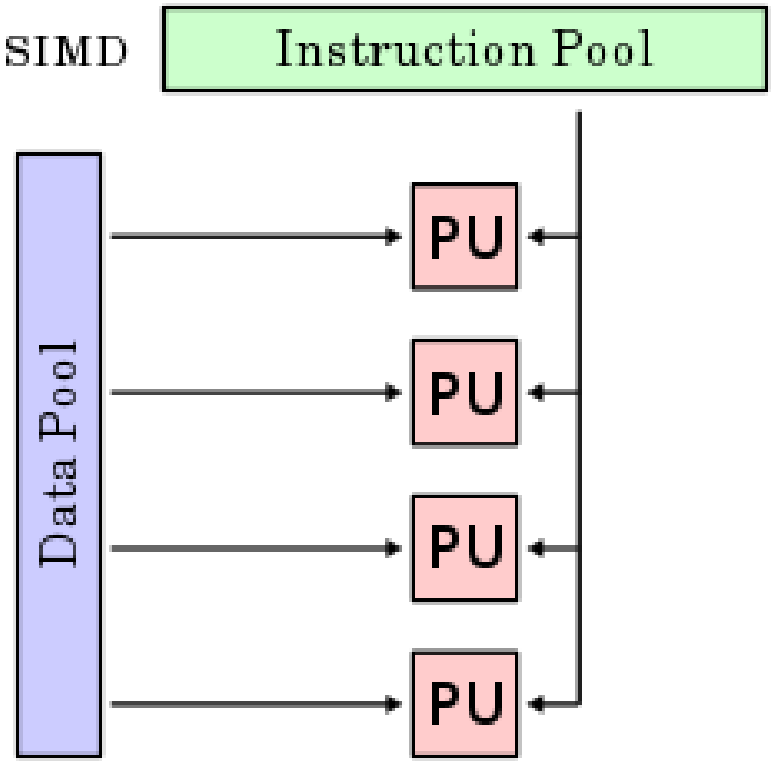
- Single Instruction, Single Data stream (SISD)
  - Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

Alternative Kinds of Parallelism: Multiple Instruction/Single Data Stream



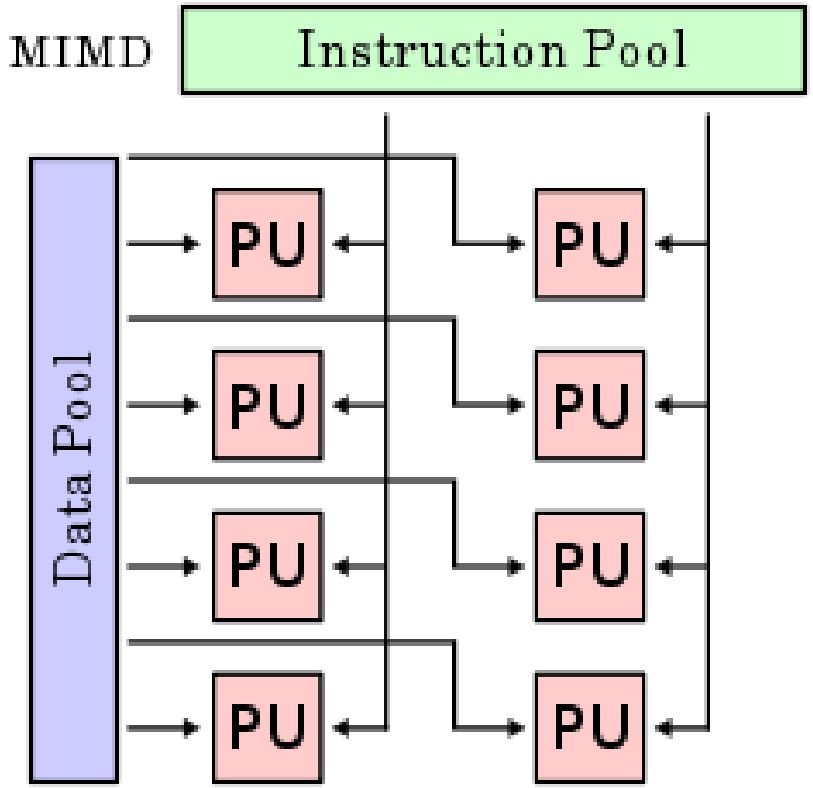
- Multiple Instruction, Single Data streams (MISD)
  - Computer that exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized. For example, certain kinds of array processors.
  - No longer commonly encountered, mainly of historical interest only

Alternative Kinds of Parallelism: Single Instruction/Multiple Data Stream



- Single Instruction, Multiple Data streams (SIMD)
  - Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., SIMD instruction extensions or Graphics Processing Unit (GPU)

Alternative Kinds of Parallelism: Multiple Instruction/Multiple Data Streams



- Multiple Instruction, Multiple Data streams (MIMD)
  - Multiple autonomous processors simultaneously executing different instructions on different data.
  - MIMD architectures include multicore and Warehouse Scale Computers (datacenters)

Flynn Taxonomy of parallel computers

		Data streams	
		Single	Parallel
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE x86, ARM neon, GPGPUs, …
	Multiple	**MISD**: No examples today	**MIMD**: SMP (Intel, ARM, …)

- **From 2011, SIMD and MIMD most common parallel computers**
- **Most common parallel processing programming style: Single Program Multiple Data ("SPMD")**
  - Single program that runs on all processors of an MIMD
  - Cross-processor execution coordination through conditional expressions (thread parallelism)
- **SIMD (aka hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays**
  - Scientific computing, signal processing, multimedia (audio/video processing)

SIMD Architectures

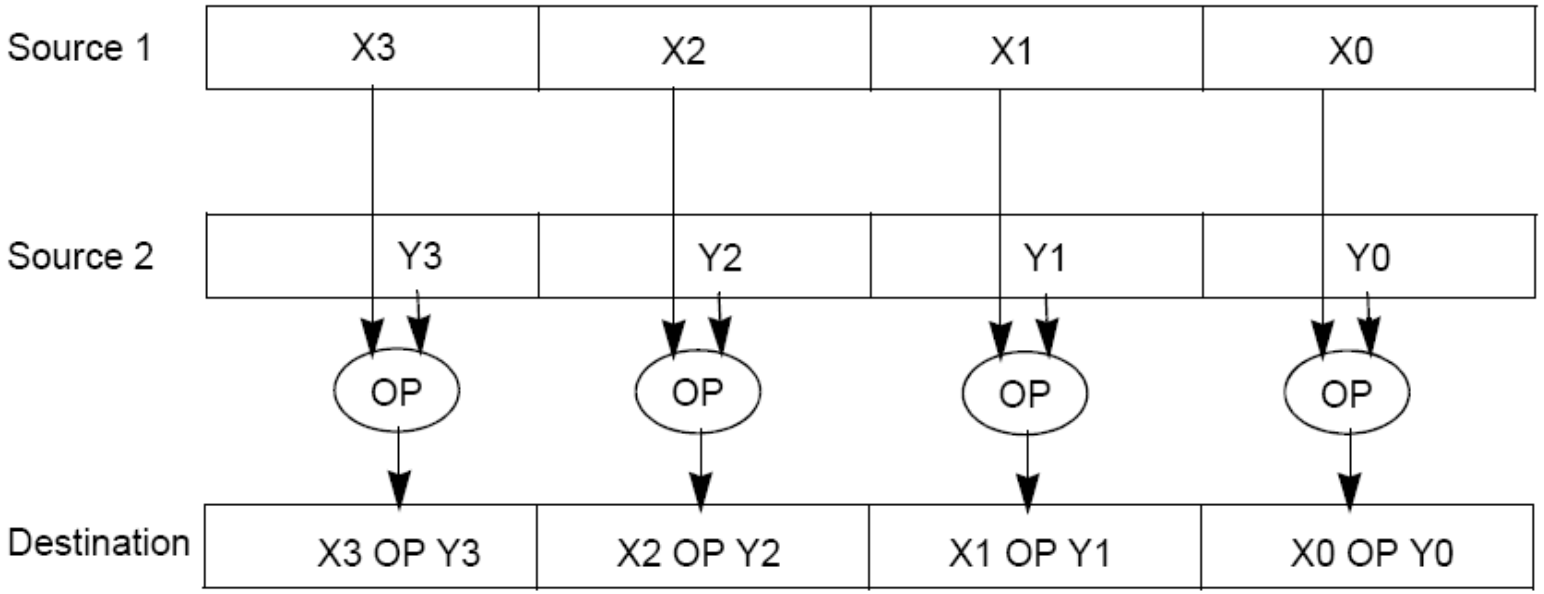
- *Data parallelism*: executing **one operation on multiple data streams**
  - Single control unit
  - Multiple datapaths (processing elements – PEs) running in parallel
    - *PEs are interconnected and exchange/share data as directed by the control unit*
    - *Each PE performs the same operation on its own local data*

- Example to provide context:
  - Multiplying a coefficient vector by a data vector (e.g., in filtering)

$$y[i] := c[i] \times x[i],\ 0 \leq i < n$$

“Advanced Digital Media Boost”

- To improve performance, SIMD instructions
  - Fetch one instruction, do the work of multiple instructions



Example: SIMD Array Processing

```
for each f in array
    f = sqrt(f)
```

```
for each f in array
{
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}
```

SISD

```
for each 4 members in array
{
    load 4 members to the SIMD register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
```

SIMD

Data Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

```
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- How can reveal more data level parallelism than available in a single iteration of a loop?
- *Unroll loop* and adjust iteration rate

Loop Unrolling

Loop Unrolling can be implemented from C code

```
for(i=1000; i>0; i=i-1)
 x[i] = x[i] + s;
```

into

```
for(i=1000; i>0; i=i-4)
{
  x[ i ] = x[ i ] + s;
  x[i-1] = x[i-1] + s;
  x[i-2] = x[i-2] + s;
  x[i-3] = x[i-3] + s;
}
```

Loop Unrolling (MIPS)

Assumptions:

- R1 is initially the address of the element in the array with the highest address
- F2 contains the scalar value s
- 8(R2) is the address of the last element to operate on.

```
for(i=1000; i>0; i=i-1)
  x[i] = x[i] + s;
```

```
Loop:
1. l.d      F0, 0(R1)     ; F0=array element
2. add.d    F4,F0,F2      ; add s to F0
3. s.d      F4,0(R1)      ; store result
4. addui    R1,R1,#-8     ; decrement pointer 8 byte
5. bne      R1,R2,Loop    ; repeat loop if R1 != R2
```

Loop Unrolled

```
Loop:  l.d     F0,0(R1)
       add.d   F4,F0,F2
       s.d     F4,0(R1)
       l.d     F6,-8(R1)
       add.d   F8,F6,F2
       s.d     F8,-8(R1)
       l.d     F10,-16(R1)
       add.d   F12,F10,F2
       s.d     F12,-16(R1)
       l.d     F14,-24(R1)
       add.d   F16,F14,F2
       s.d     F16,-24(R1)
       addui   R1,R1,#-32
       bne     R1,R2,Loop
```

NOTE:

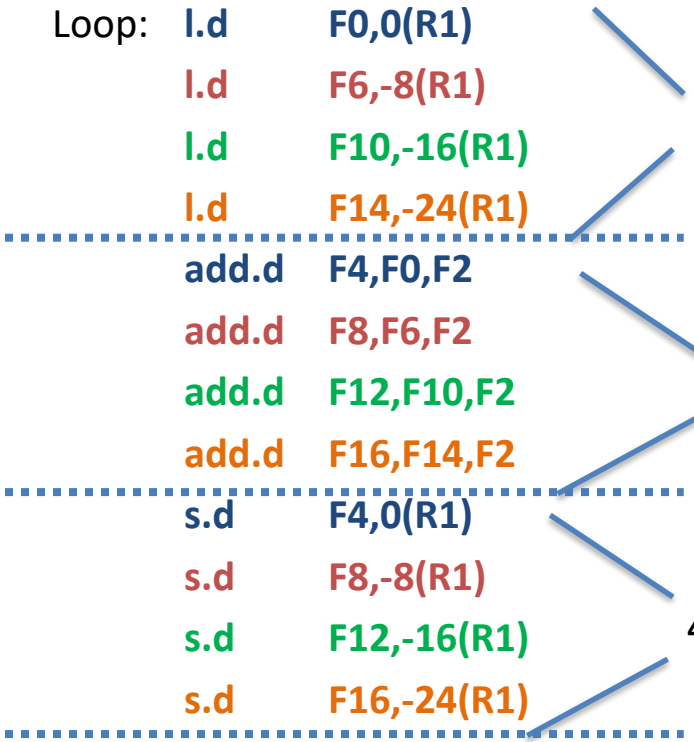
1. Different Registers eliminate stalls
2. Only 1 Loop Overhead every 4 iterations
3. This unrolling works if loop_limit(mod 4) = 0

Loop Unrolled Scheduled

```
Loop:  l.d     F0,0(R1)
       l.d     F6,-8(R1)
       l.d     F10,-16(R1)
       l.d     F14,-24(R1)
       add.d   F4,F0,F2
       add.d   F8,F6,F2
       add.d   F12,F10,F2
       add.d   F16,F14,F2
       s.d     F4,0(R1)
       s.d     F8,-8(R1)
       s.d     F12,-16(R1)
       s.d     F16,-24(R1)
       addui   R1,R1,#-32
       bne     R1,R2,Loop
```

Loop Unrolled Scheduled

```
Loop:  l.d     F0,0(R1)
       l.d     F6,-8(R1)
       l.d     F10,-16(R1)
       l.d     F14,-24(R1)
       add.d   F4,F0,F2
       add.d   F8,F6,F2
       add.d   F12,F10,F2
       add.d   F16,F14,F2
       s.d     F4,0(R1)
       s.d     F8,-8(R1)
       s.d     F12,-16(R1)
       s.d     F16,-24(R1)
       addui   R1,R1,#-32
       bne     R1,R2,Loop
```



4 Loads side-by-side: Could replace with 4 wide SIMD Load

4 Adds side-by-side: Could replace with 4 wide SIMD Add

4 Stores side-by-side: Could replace with 4 wide SIMD Store

Generalizing Loop Unrolling

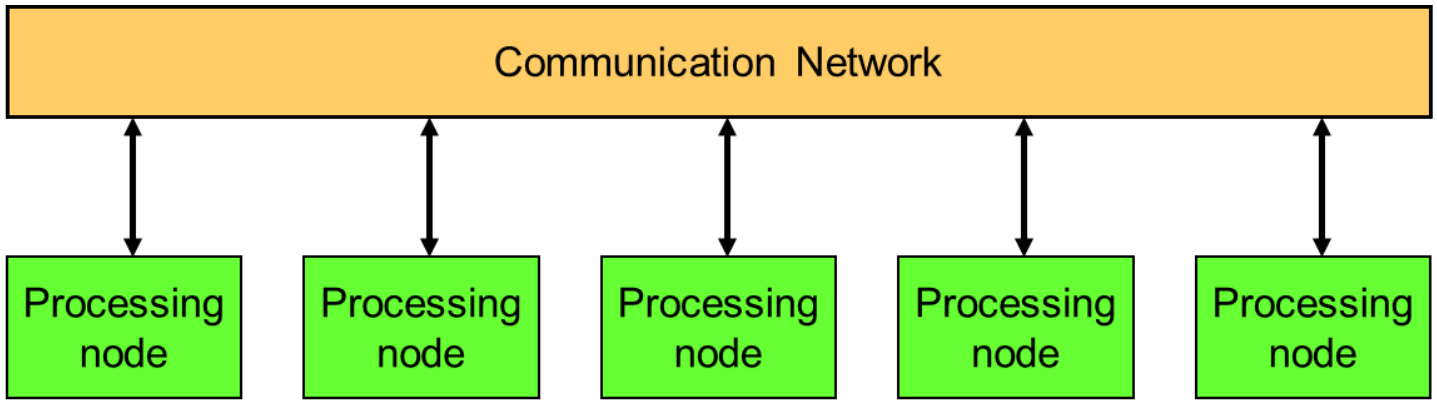
- A loop of **n iterations**
- **k copies** of the body of the loop

Then we will run the loop with 1 copy of the body **n(mod k)** times and

with k copies of the body **floor(n/k)** times

MIMD Architectures

- **Multicore architectures**
- At least 2 processors interconnected via a communication network
  - abstractions (HW/SW interface)
  - organizational structure to realize abstraction efficiently



MIMD Architectures

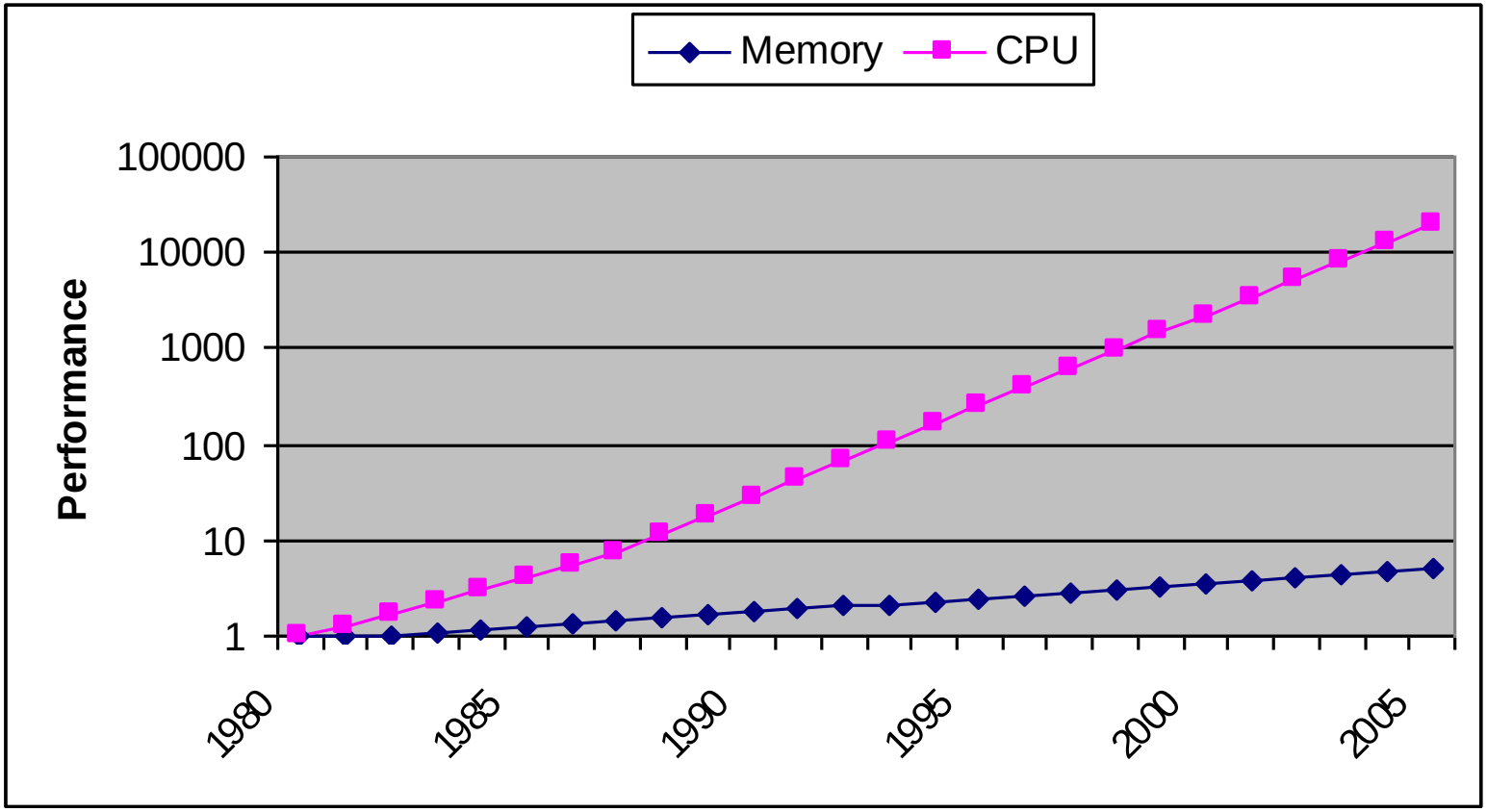
- Thread-Level parallelism
  - Have multiple program counters
  - Targeted for tightly-coupled shared-memory multiprocessors
- For *n* processors, need *n* threads
- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

MIMD Architectures

- And what about memory?
- How is data to be accessed by multiple cores?
- How to design accordingly the memory system?
- How do traditional solutions from uniprocessor systems affect multicores?

Memory
and communication paradigms

Recall: The Memory Gap



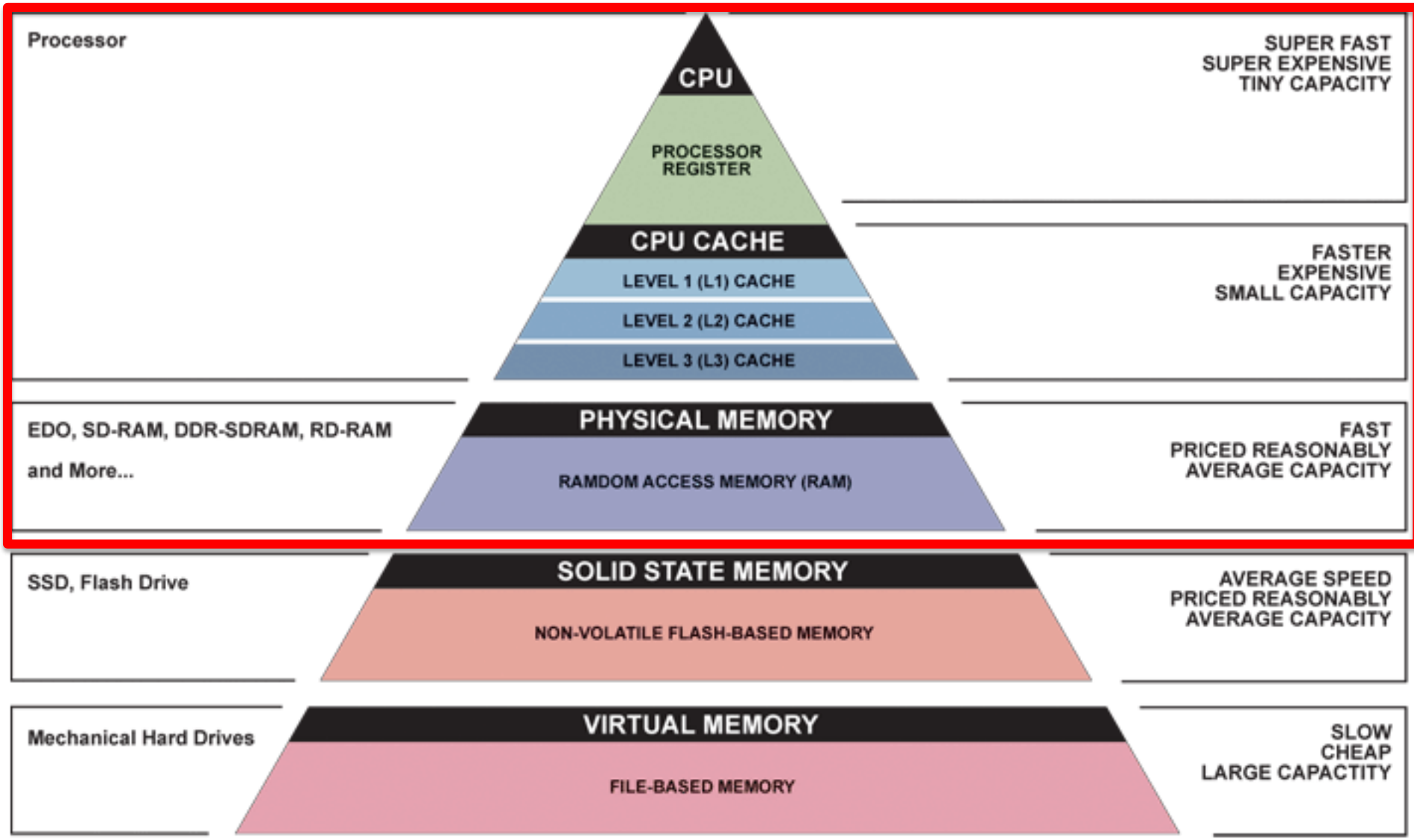
H&P
Fig. 5.2

- **Bottom-line: memory access is increasingly expensive and CA must devise new ways of hiding this cost**

The memory wall

- How do multicores attach the memory wall?
- Same issue as uniprocessor systems
  - By building on-chip cache hierarchies

Memory Hierarchy



The memory wall

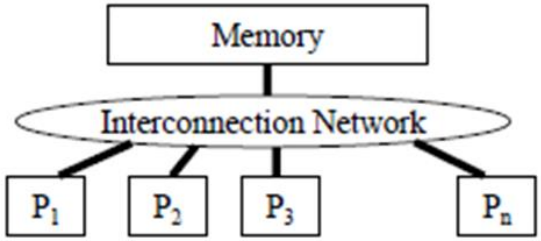
- How do multicores attach the memory wall?
- Same issue as uniprocessor systems
  - By building on-chip cache hierarchies

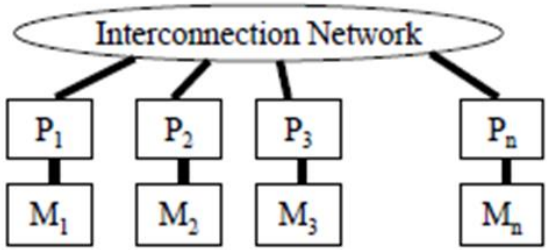
- Two main paradigms
  - Shared memory
  - Distributed memory

- What novel issues arise?

Multicore design paradigm

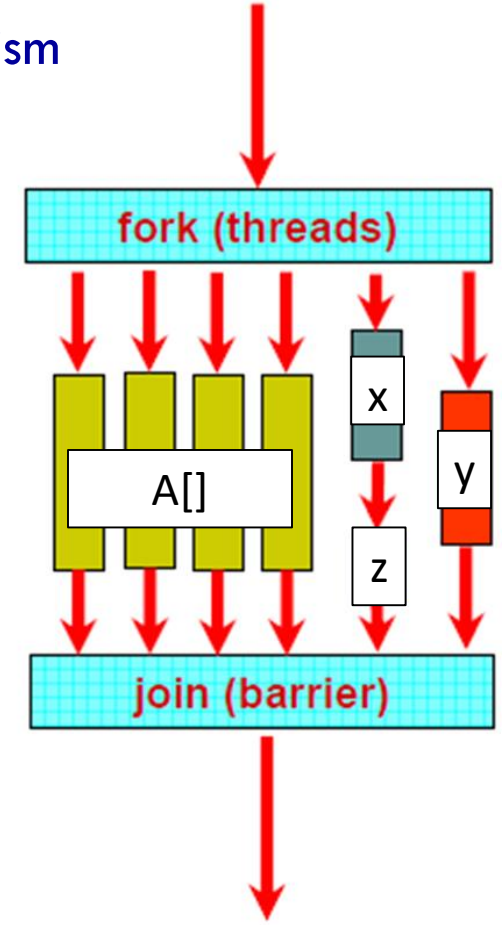
- Two primary patterns of multicore architecture design
  - Shared memory
    - One copy of data shared among many cores
    - Atomicity, locking and synchronization essential for correctness
    - Many scalability issues
  - Distributed memory
    - Cores primarily access local memory
    - Explicit data exchange between cores
    - Data distribution and communication orchestration is essential for performance





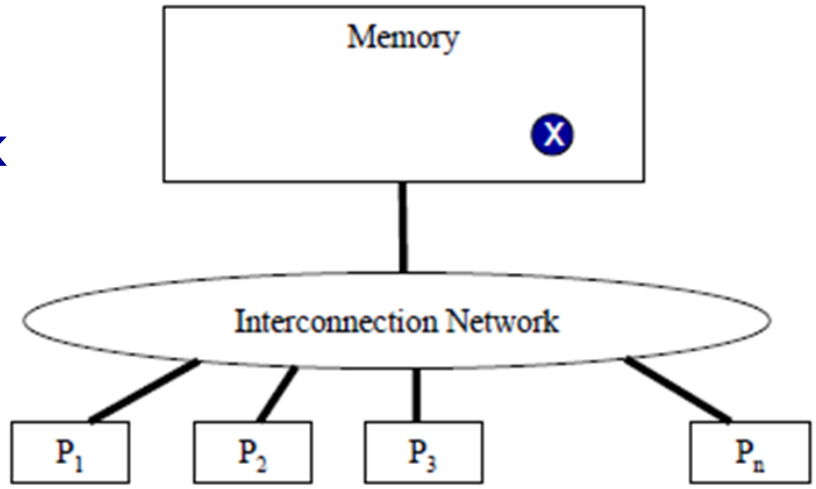
Types of Parallelism

- And, orthogonal to that, two main types of parallelism
- Data parallelism
  - Perform same computation
    but operate on different data
- Task (control) parallelism
  - Perform different functions



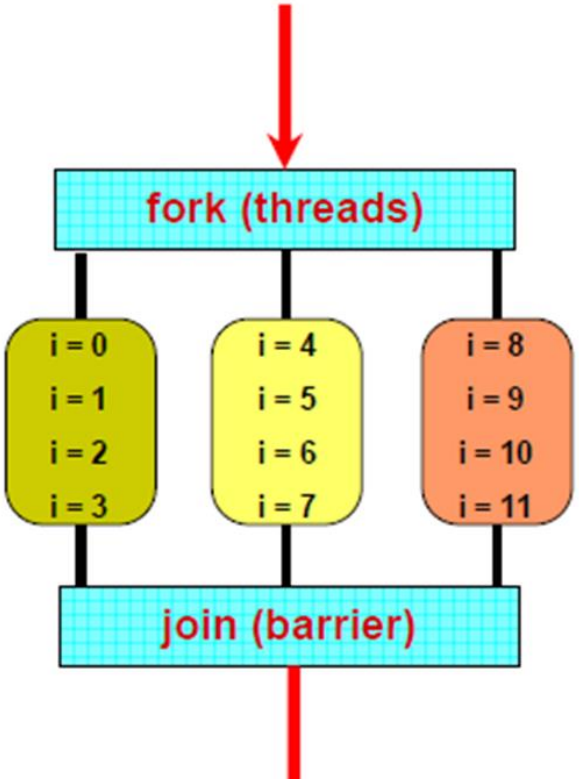
Shared Memory Programming

- Processor 1..n ask for X
- There is only one place to look
- Communication through **shared variables**



Example

```
for (i = 0; i < 12; i++)
      C[i] = A[i] + B[i];
```



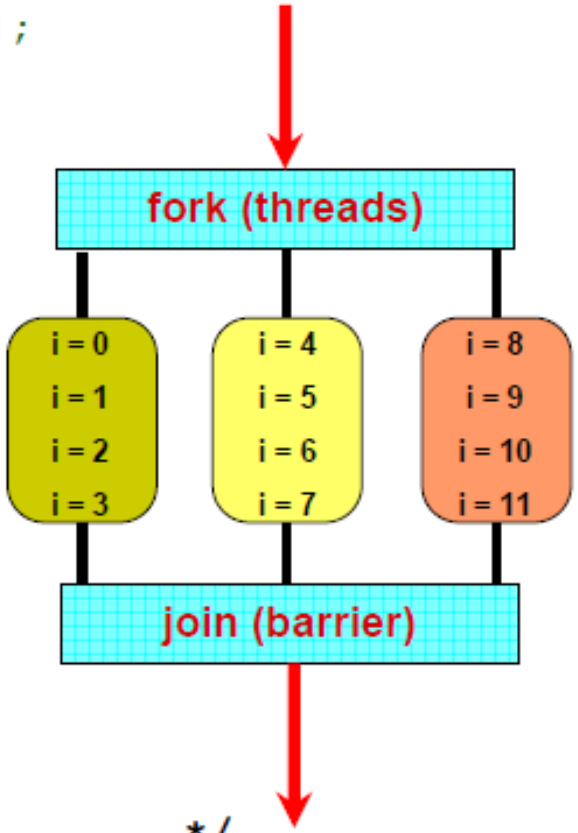
- Data parallel
  - Perform same computation but operate on different data
- A single process can fork multiple concurrent threads
  - Each thread encapsulates its own execution path
  - Each thread has local state and shared resources
  - Threads communicate through shared resources such as global memory

Example: *pthreads*

```
int A[12] = {...}; int B[12] = {...}; int C[12];

void add_arrays(int start)
{
   int i;
   for (i = start; i < start + 4; i++)
     C[i] = A[i] + B[i];
}

int main (int argc, char *argv[])
{
   pthread_t threads_ids[3];
   int rc, t;
   for(t = 0; t < 4; t++) {
     rc = pthread_create(&thread_ids[t],
                         NULL        /* attributes       */,
                         add_arrays  /* function         */,
                         t * 4       /* args to function */);
   }
   pthread_exit(NULL);
}
```



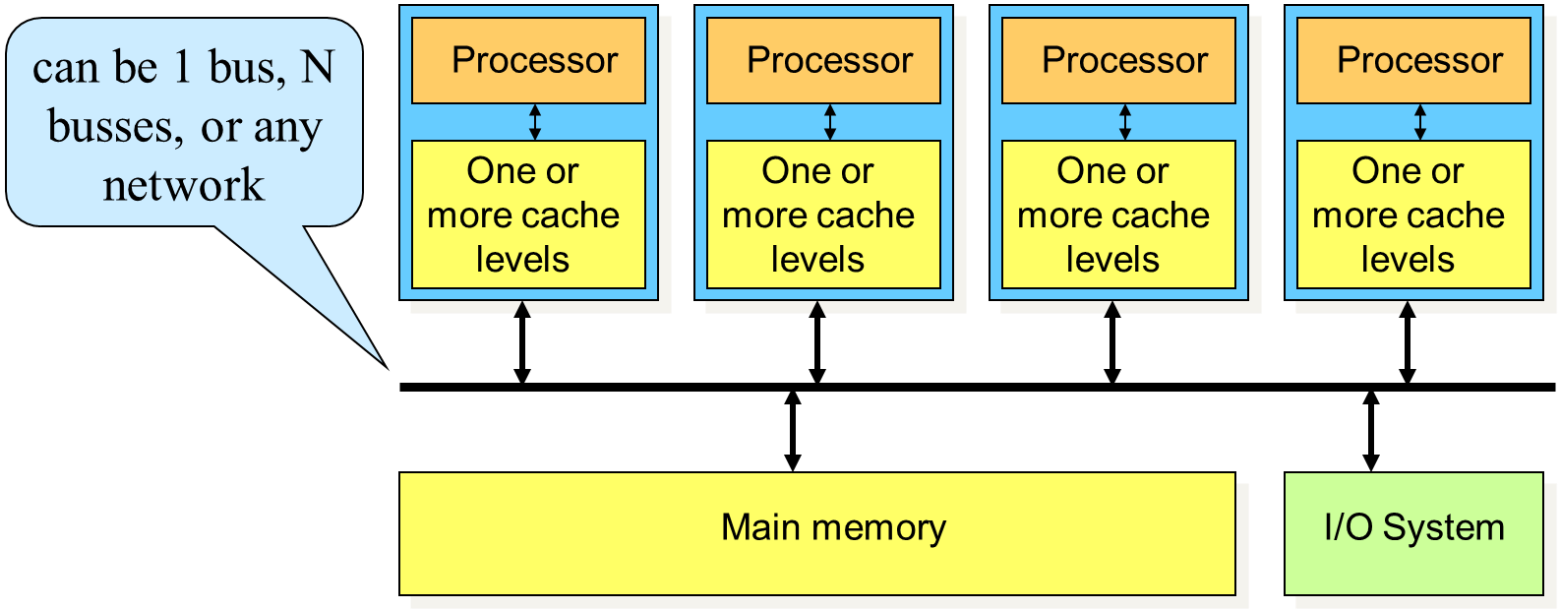
Shared memory: physical implementation

Two varieties:

- Physically shared => Symmetric Multi-Processors (**SMP**)
  - usually combined with local caching
- Physically distributed => Distributed Shared Memory (**DSM**)

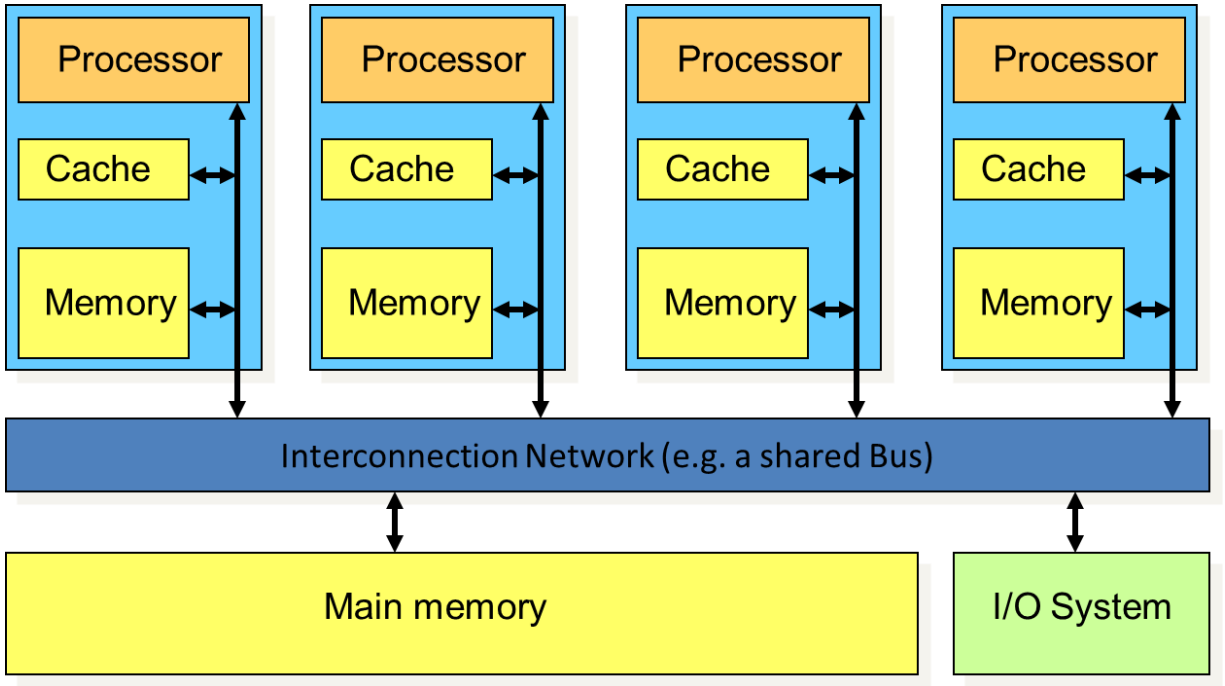
SMP: Symmetric Multi-Processor

- Memory: centralized with uniform access time (***UMA***) and bus interconnect, I/O
- Examples: Intel, ARM A-series



DSM: Distributed Shared Memory

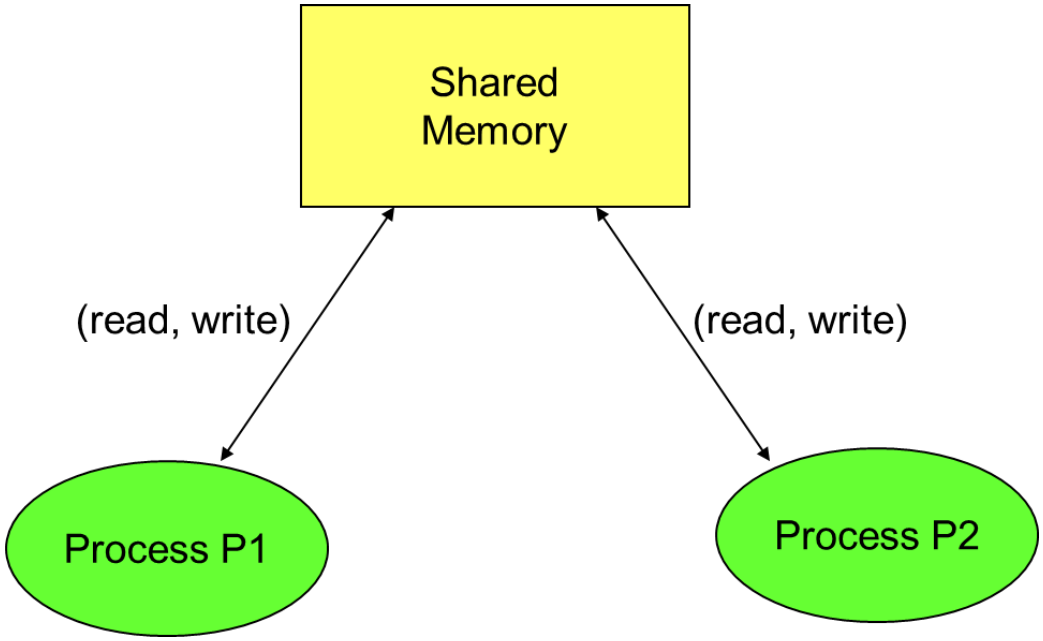
- Non-uniform access time (***NUMA***) and scalable interconnect (distributed memory)



Shared Address Model Summary

- Each processor can address every physical location in the machine
- Each process can address all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Memory hierarchy model applies:
  - communication moves data to local proc. cache

Shared Memory: novel issues



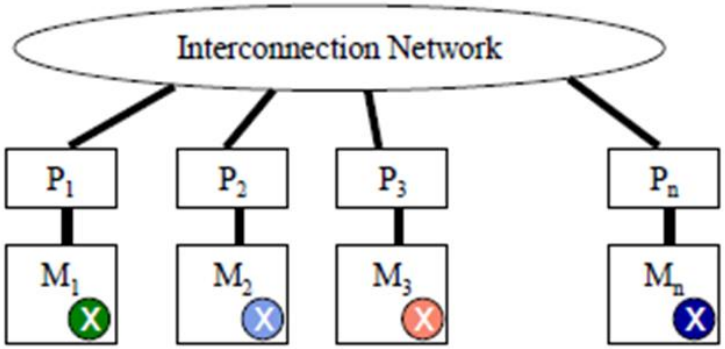
- Coherence problem
- Memory consistency issue
- Synchronization problem

- Shared address space
- Communication primitives:
  - load, store, atomic swap

we'll cover these in a while…

Distributed Memory Programming

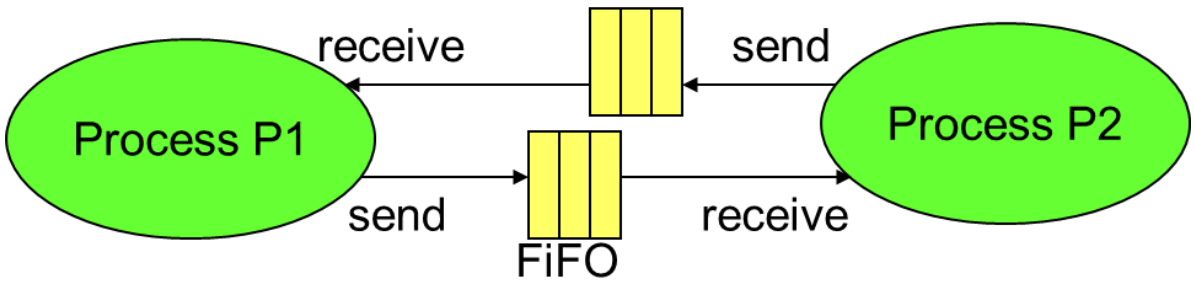
- Processor 1..n ask for X
- There are n places to look
  - Each processor's memory has its own X
  - Xs may vary



- For Processor 1 to look at Processor's 2's X
  - Processor 1 has to request X from Processor 2
  - Processor 2 sends a copy of its own X to Processor 1
  - Processor 1 receives the copy
  - Processor 1 stores the copy in its own memory

Communication models: **Message Passing**

- Communication primitives
  - e.g., send, receive library calls
  - standard MPI: Message Passing Interface
    - www.mpi-forum.org
- *Note that MP can be build on top of SM and vice versa !*

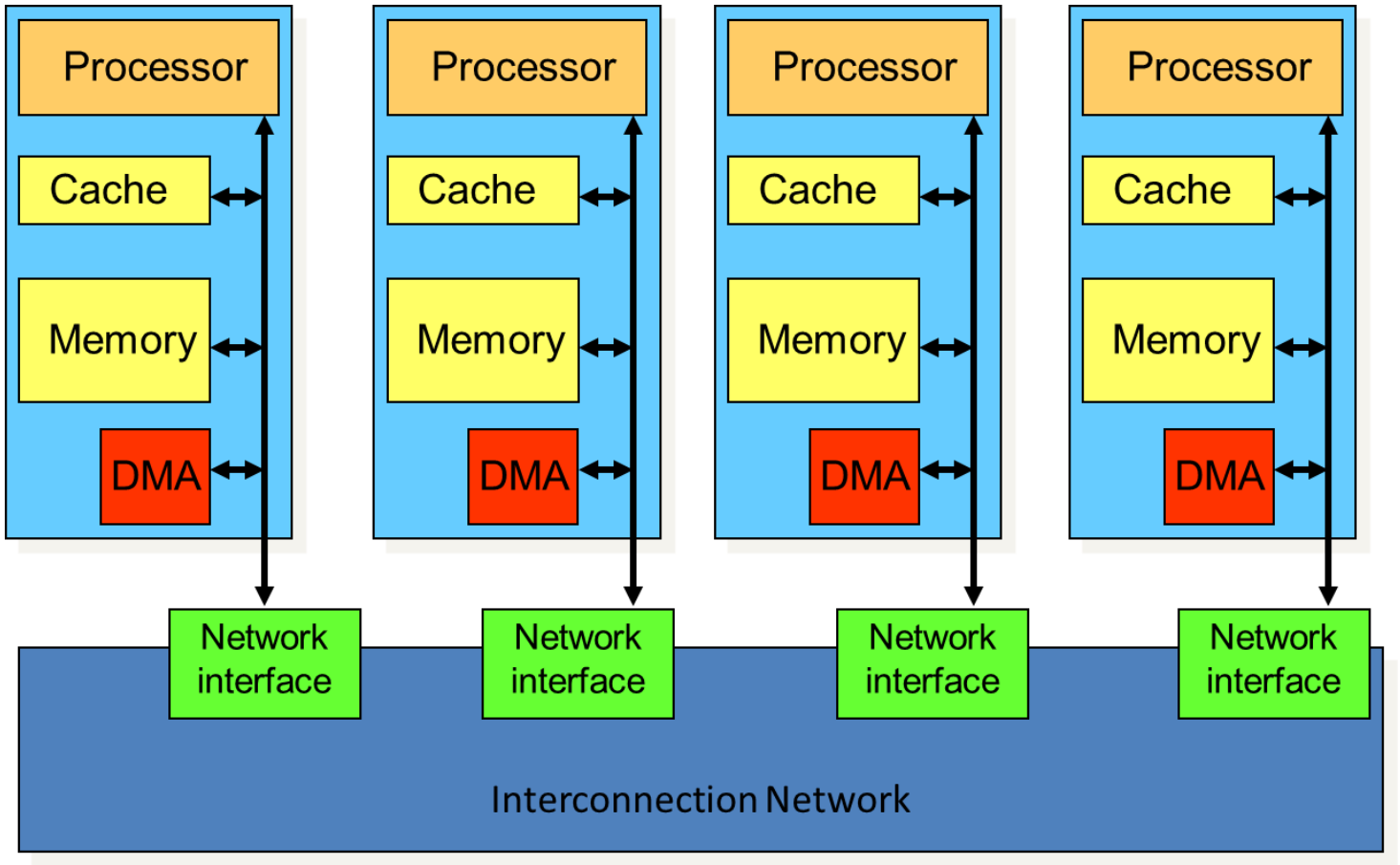


Message Passing Model

- Explicit message send and receive operations
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
- Typically blocking communication, but may use DMA

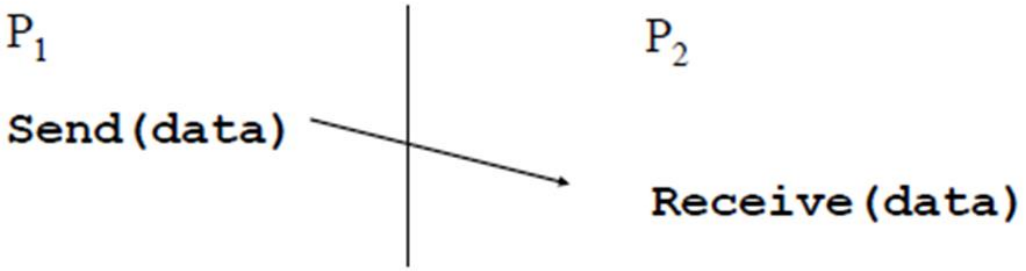
Message structure

Message passing: physical implementation



Message Passing

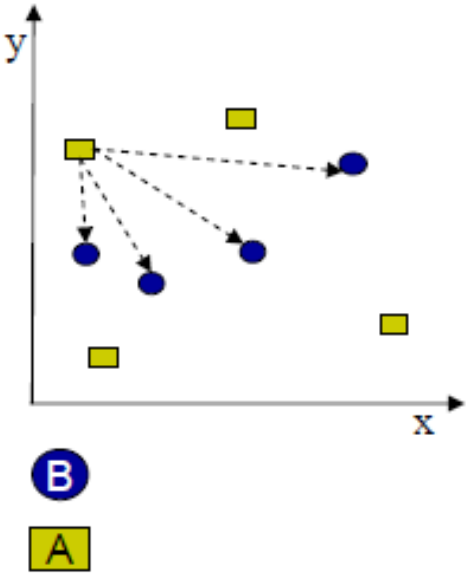
- Architectures with distributed memories use explicit communication to exchange data
  - Data exchange requires synchronization (cooperation) between senders and receivers



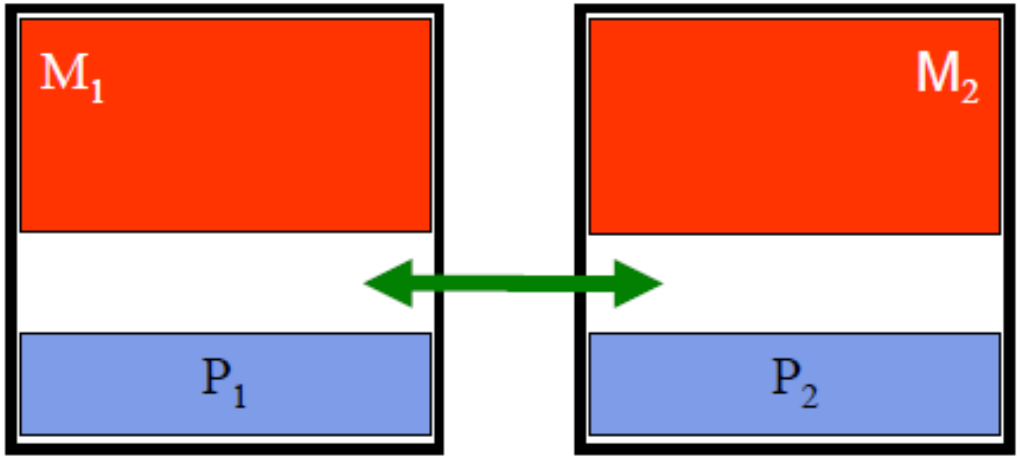
- How is “data” described
- How are processes identified
- Will receiver recognize or screen messages
- What does it mean for a send or receive to complete

Example

- Calculate the distance from each point in `A[1..4]` to every other point in `B[1..4]` and store results to `C[1..4][1..4]`

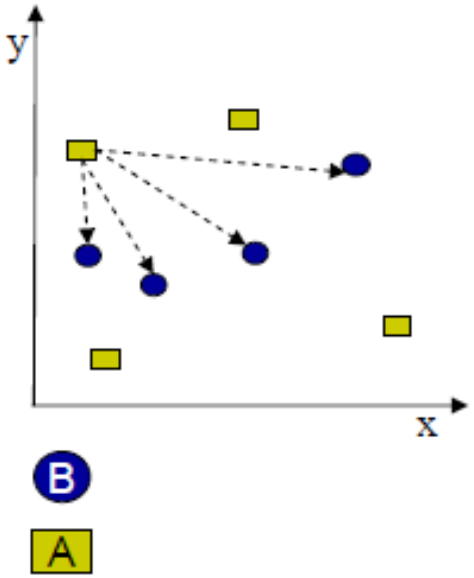



```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

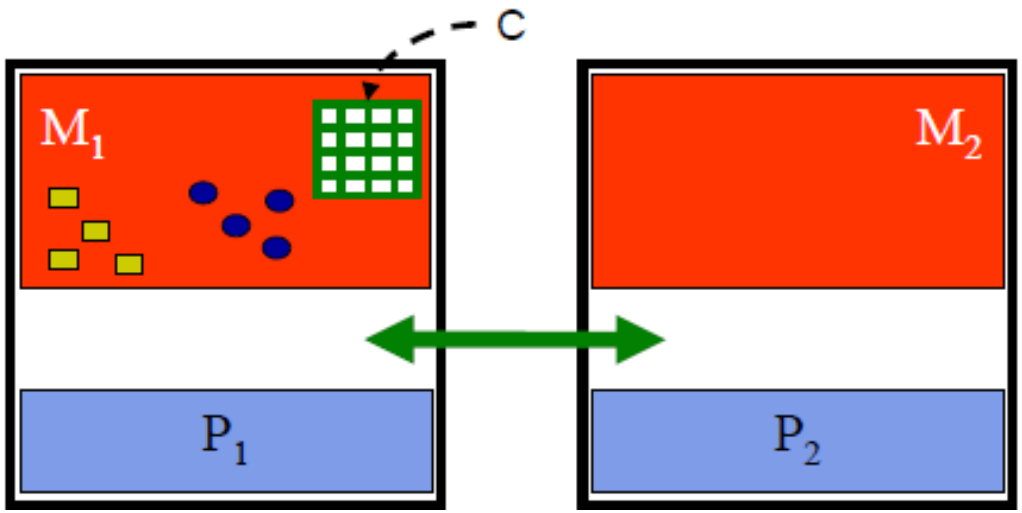


Example

- Calculate the distance from each point in **A[1..4]** to every other point in **B[1..4]** and store results to **C[1..4][1..4]**




```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

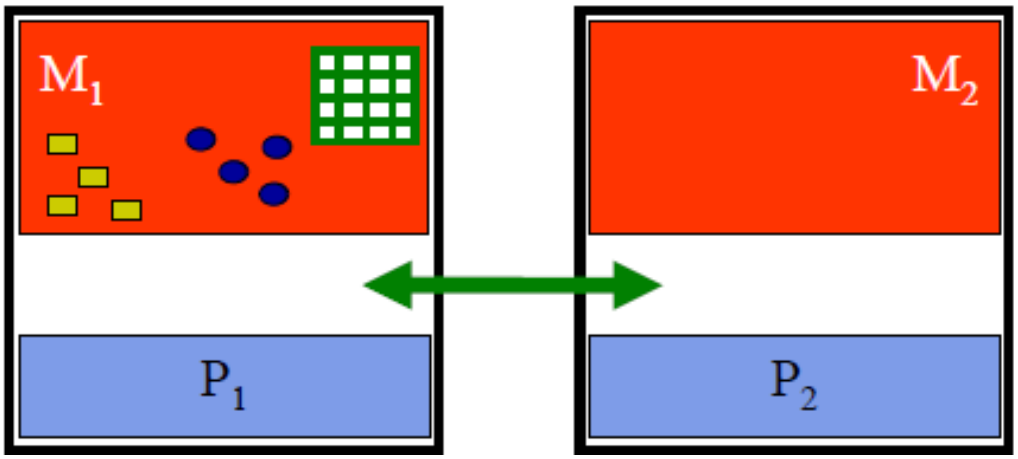


Example

- Calculate the distance from each point in `A[1..4]` to every other point in `B[1..4]` and store results to `C[1..4][1..4]`

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

- Can break up work between the two processors
  - P_1 sends data to P_2

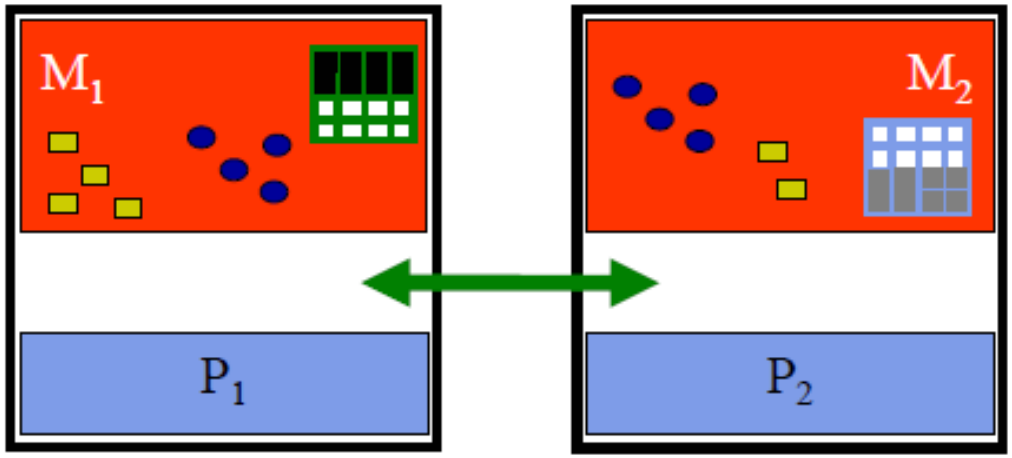


Example

- Calculate the distance from each point in `A[1..4]` to every other point in `B[1..4]` and store results to `C[1..4][1..4]`

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

- Can break up work between the two processors
  - P_1 sends data to P_2
  - P_1 and P_2 compute

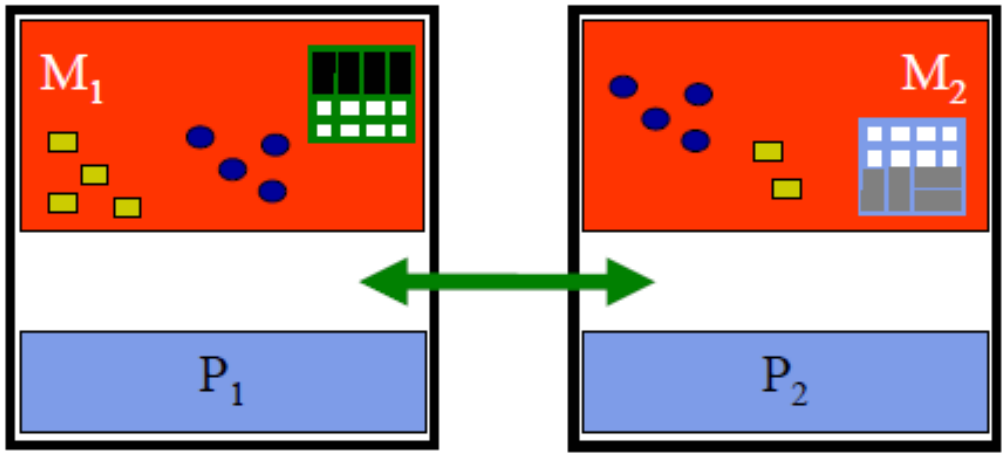


Example

- Calculate the distance from each point in `A[1..4]` to every other point in `B[1..4]` and store results to `C[1..4][1..4]`

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

- Can break up work between the two processors
  - P_1 sends data to P_2
  - P_1 and P_2 compute
  - P_2 sends output to P_1



Example

processor 1

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

sequential

parallel with messages

processor 1

```
A[n] = {…}
B[n] = {…}

Send (A[n/2+1..n], B[1..n])

for (i = 1 to n/2)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])

Receive(C[n/2+1..n][1..n])
```

processor 2

```
A[n] = {…}
B[n] = {…}

Receive(A[n/2+1..n], B[1..n])

for (i = n/2+1 to n)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])

Send (C[n/2+1..n][1..n])
```

Communication Models: Comparison

- Shared-Memory (used by e.g. **OpenMP**)
  - Compatibility with well-understood (language) mechanisms
  - Ease of programming for complex or dynamic communications patterns
  - Shared-memory applications; sharing of large data structures
  - Efficient for small items
  - Supports hardware caching

- Messaging Passing (used by e.g. **MPI**)
  - Simpler hardware
  - Explicit communication
  - Implicit synchronization (with any communication)

Novel issues
for Shared Memory Multiprocessors

Three fundamental issues for shared memory multiprocessors

- **Coherence**
  Do I see the most recent data?

- **Synchronization**
  How to synchronize processes?
  - how to protect access to shared data?

- **Consistency**
  ***When** do I see a written value?*
  - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

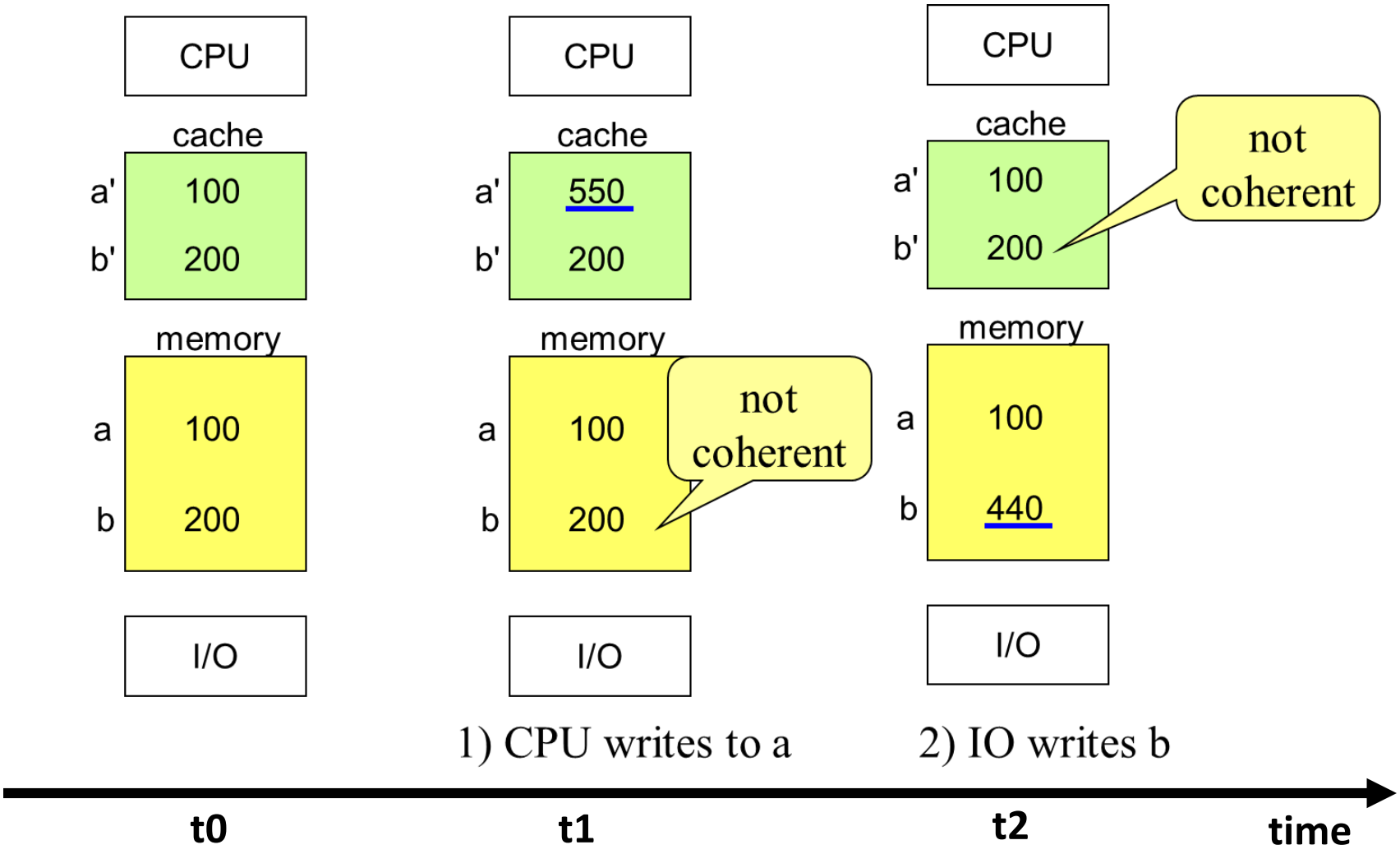
Three fundamental issues for shared memory multiprocessors

- **Coherence**
  Do I see the most recent data?

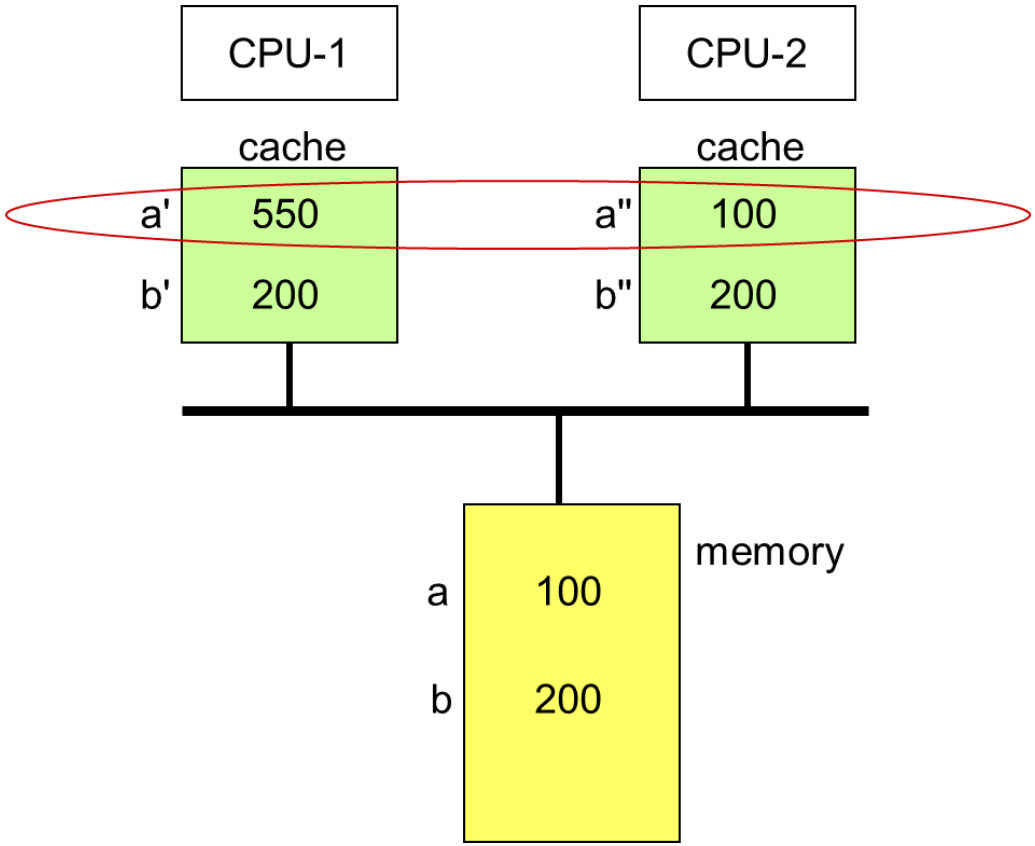
- **Synchronization**
  How to synchronize processes?
  - how to protect access to shared data?

- **Consistency**
  ***When** do I see a written value?*
  - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

Coherence problem, in single CPU system



Coherence problem, in Multi-Proc system



What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write (to the same address)"
  - Too strict and too difficult to implement

- Better:
  - A write followed by a read by the same processor P with no writes in between returns the value written
  - "Any write must eventually be seen by a read"

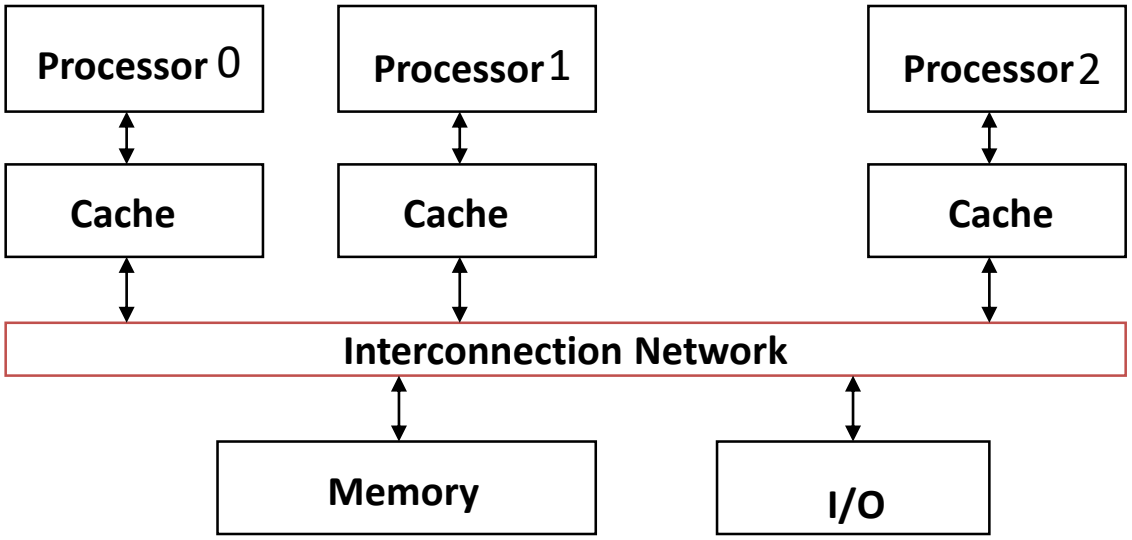
Two rules to ensure coherency

- "If P1 writes x and P2 reads it, P1's write will be seen by P2 if the read and write are sufficiently far apart"

- Writes to a single location are serialized: seen in one order
  - 'Latest' write will be seen
  - Otherwise could see writes in illogical order (could see older value after a newer value)

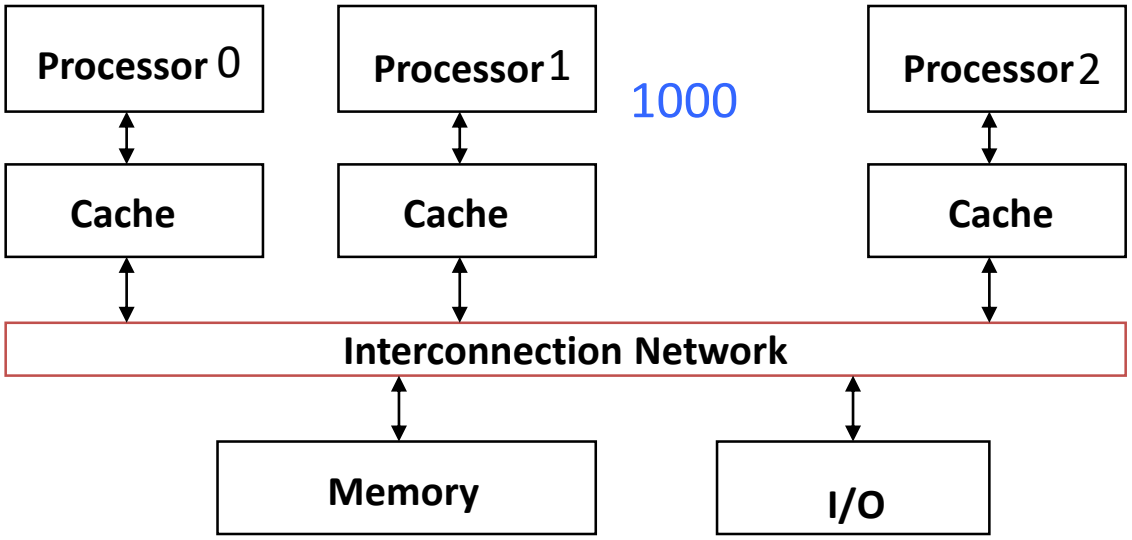
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



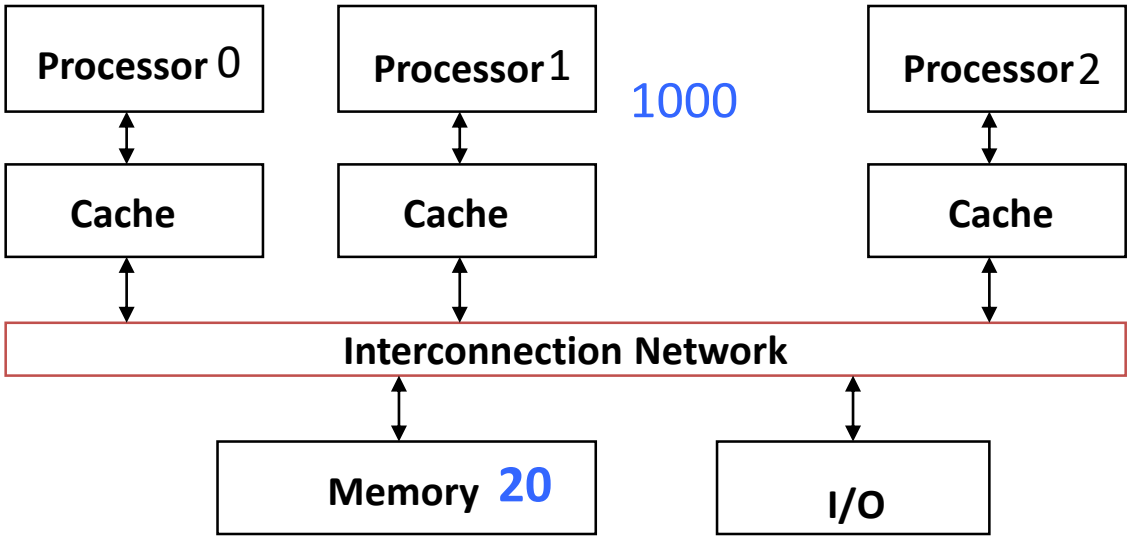
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



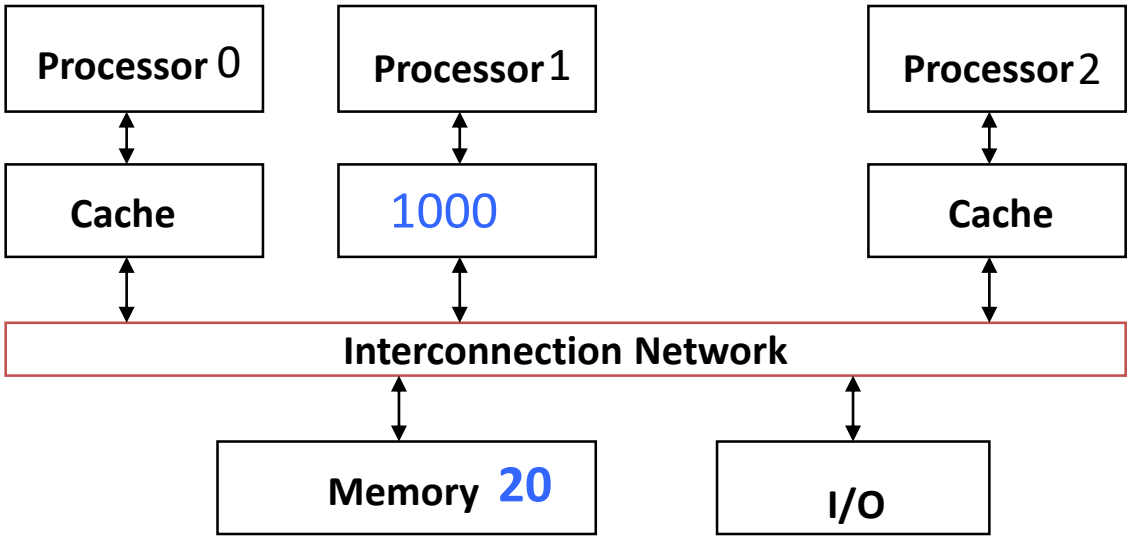
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



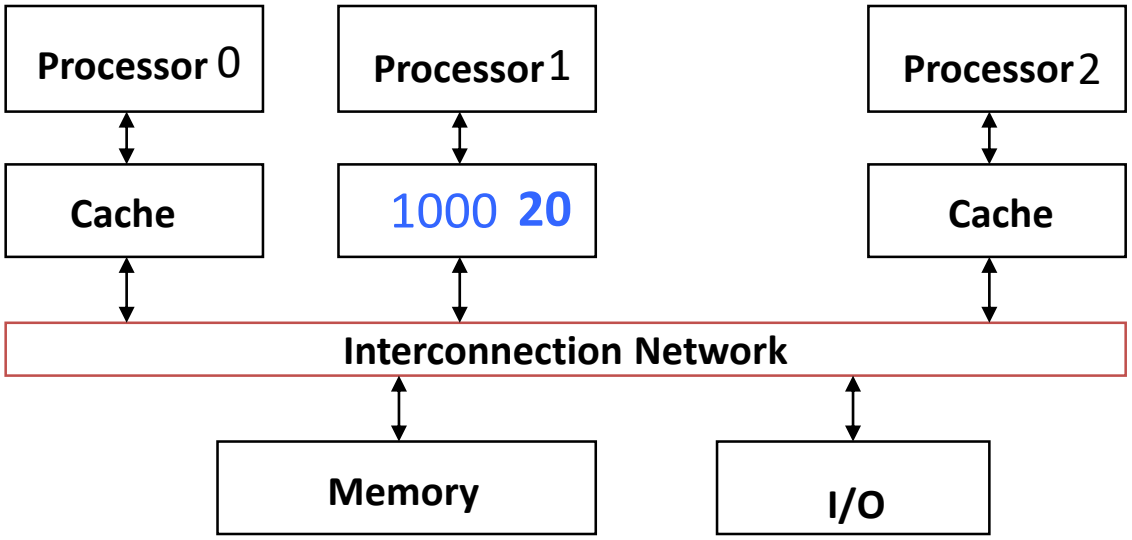
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



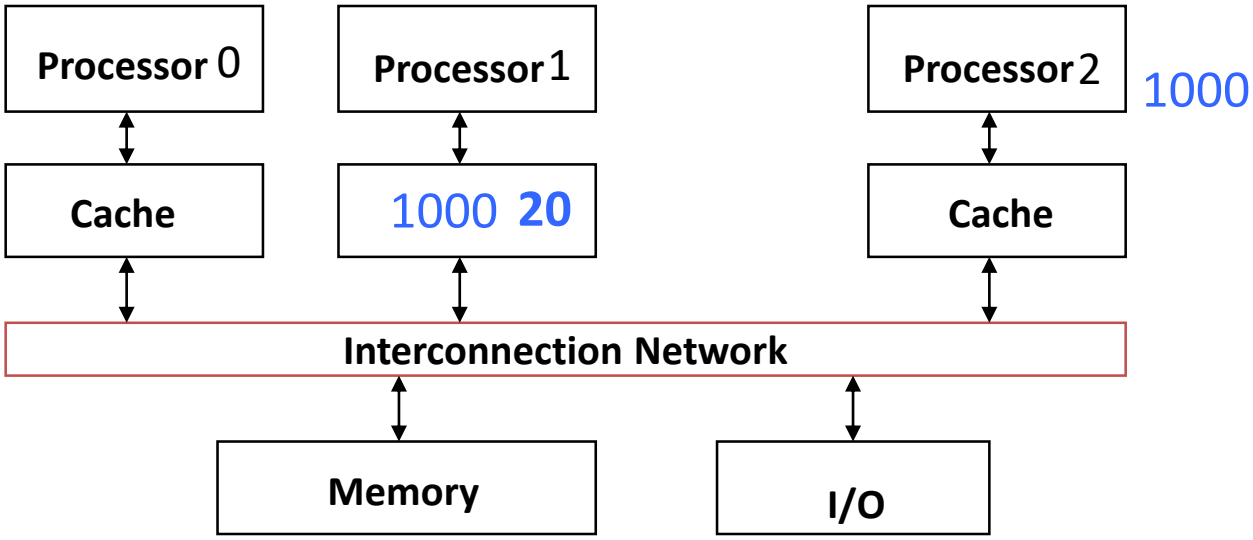
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



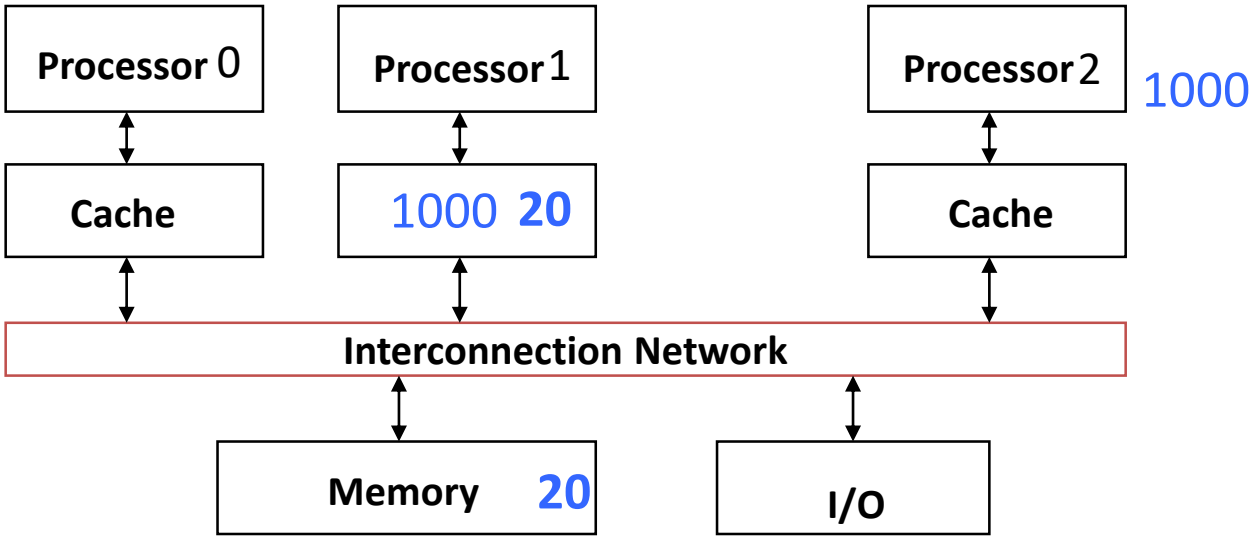
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



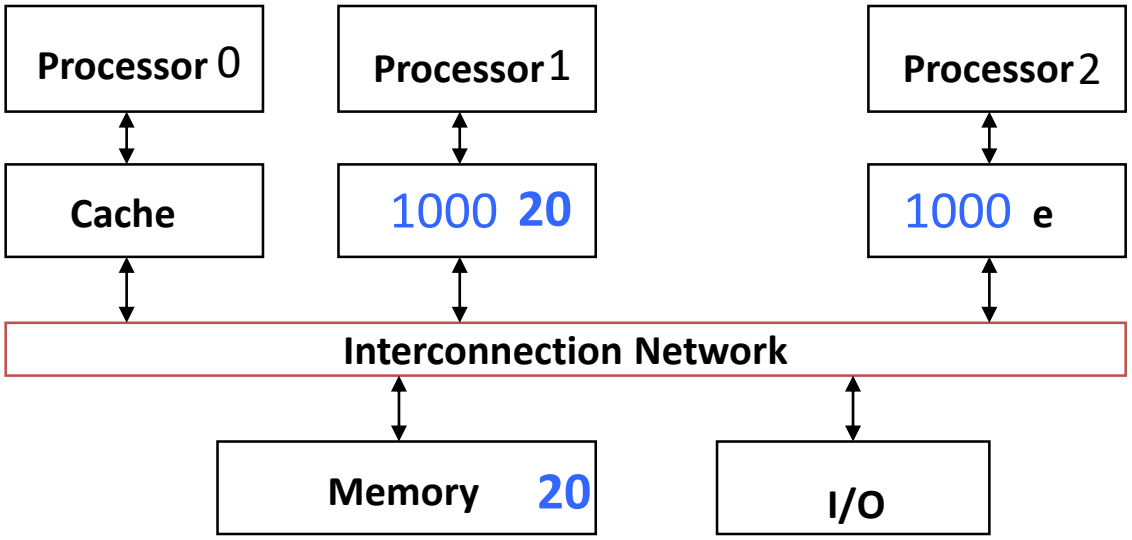
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



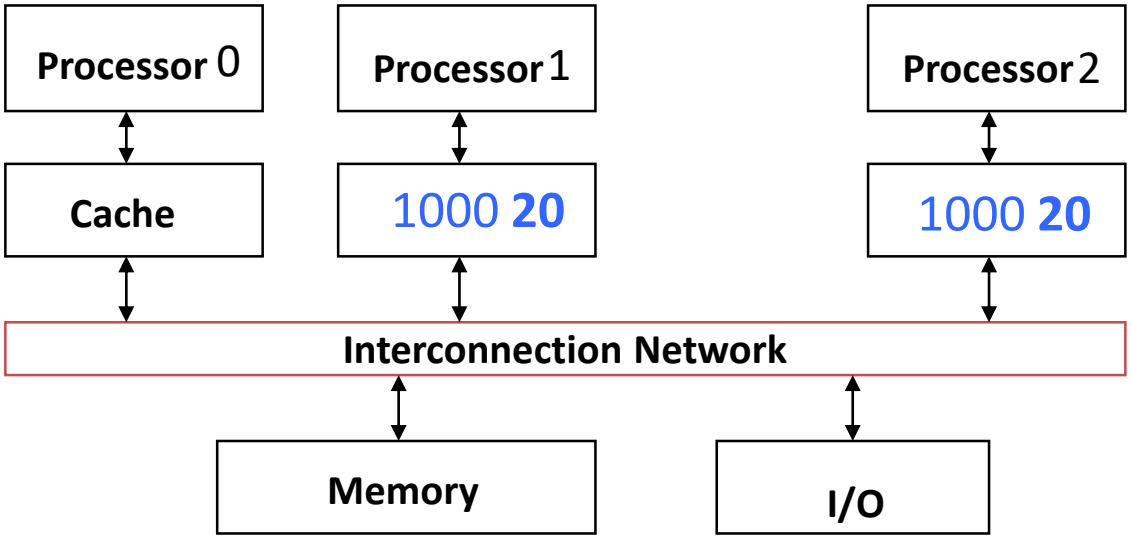
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



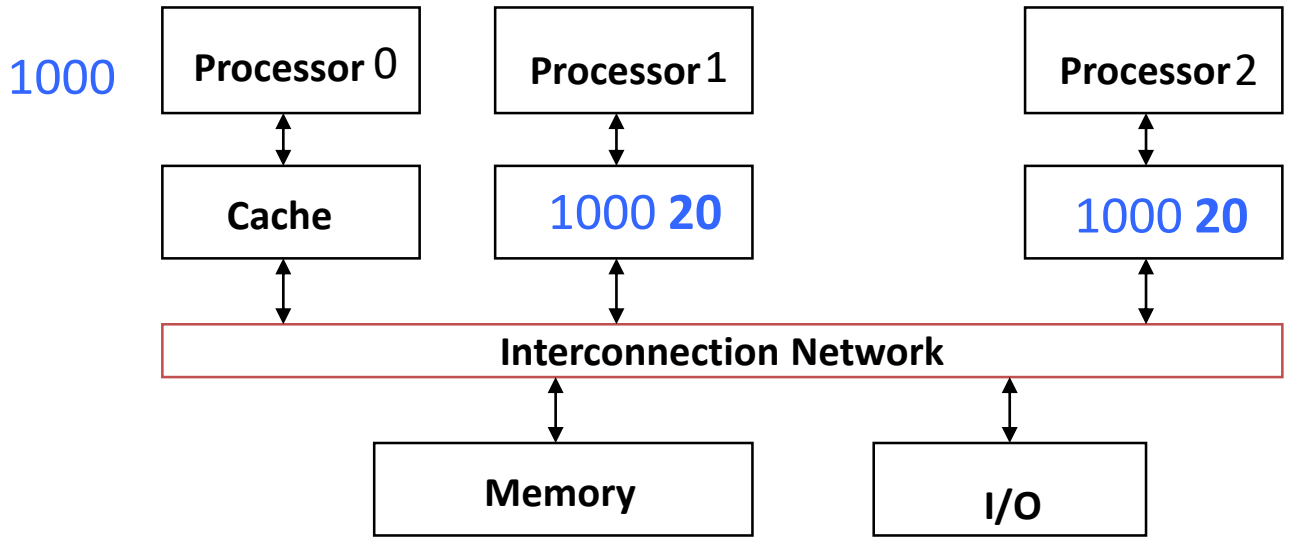
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



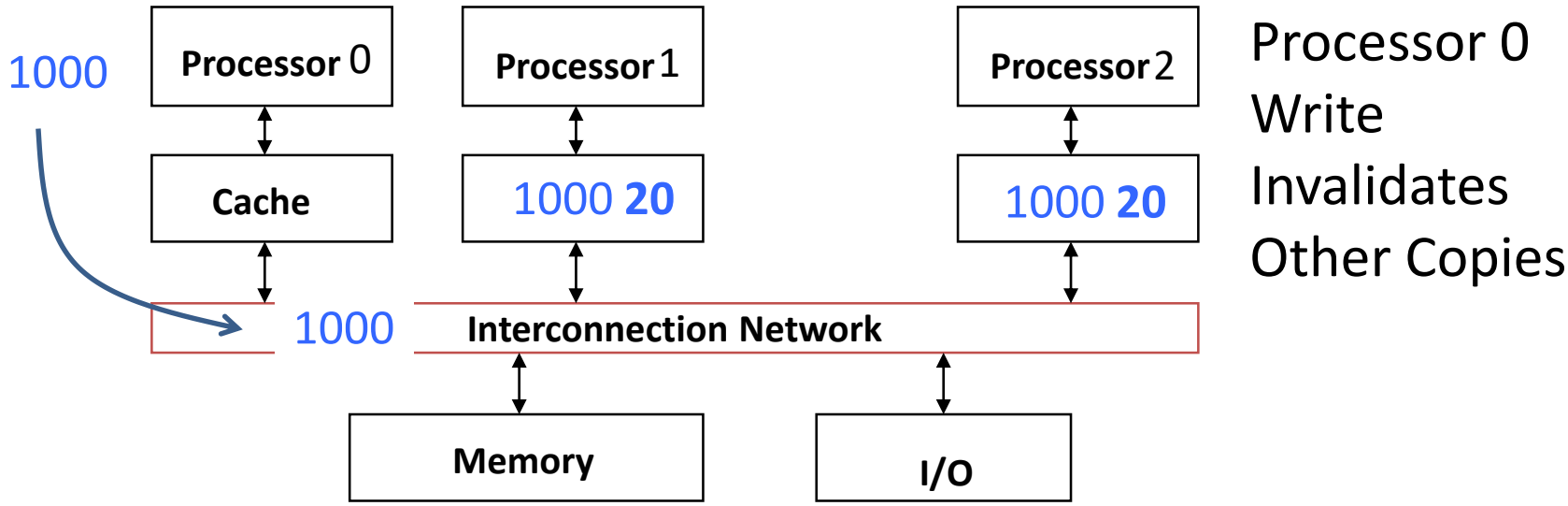
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



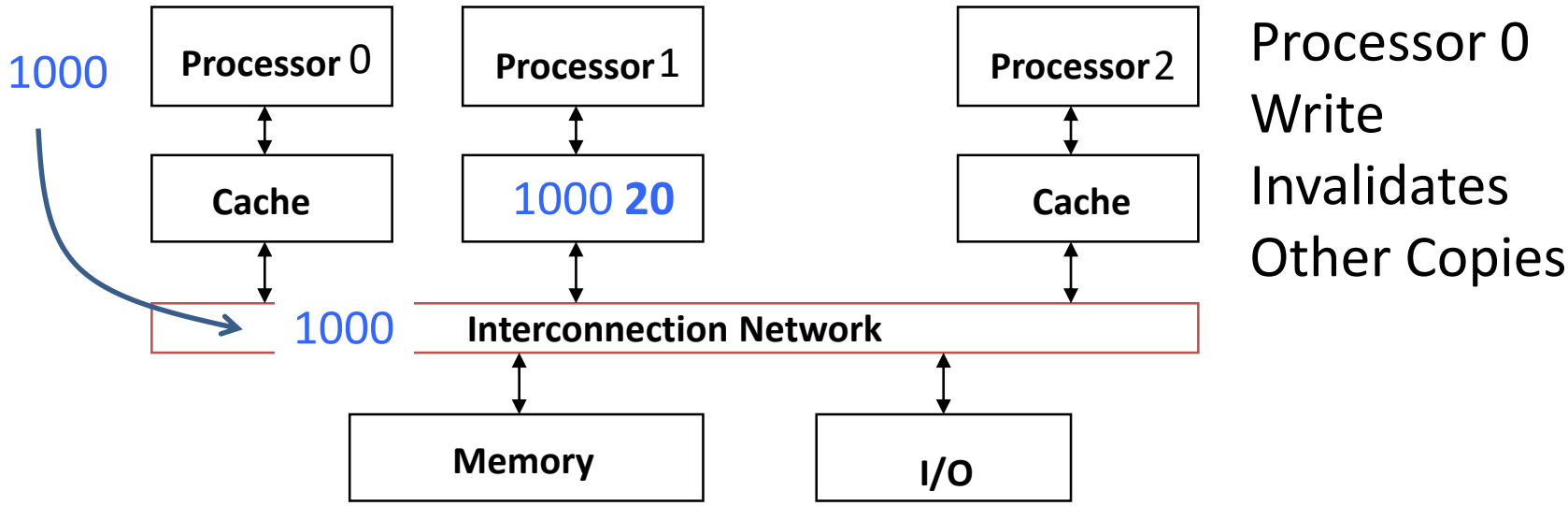
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



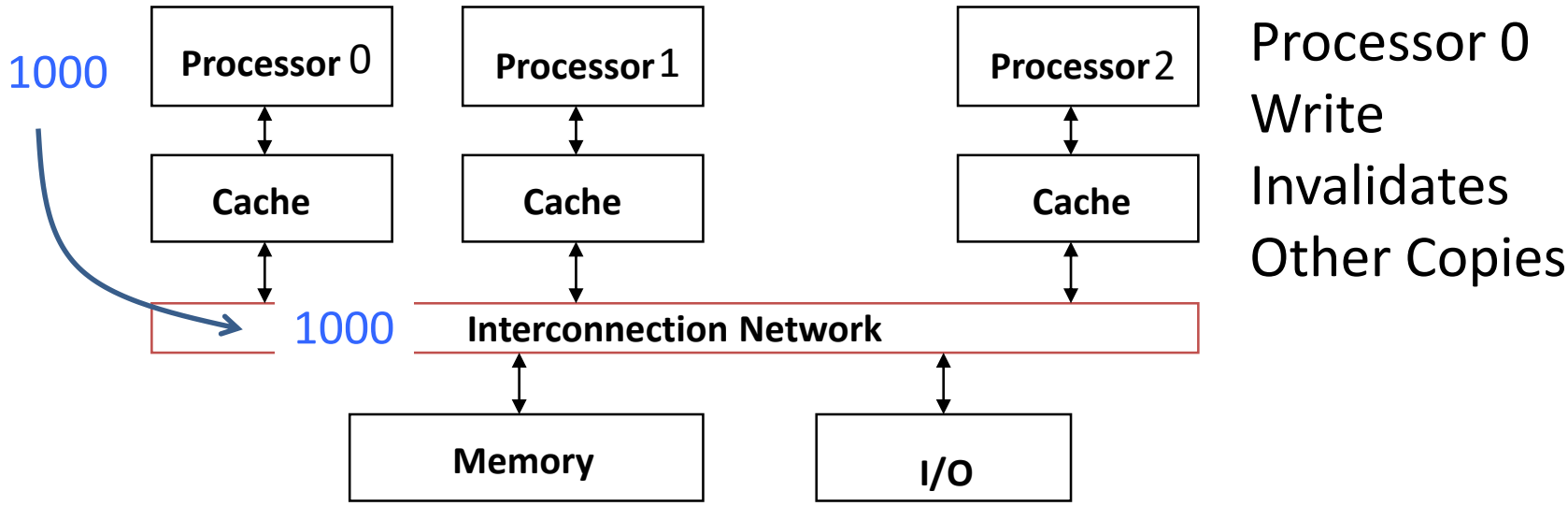
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



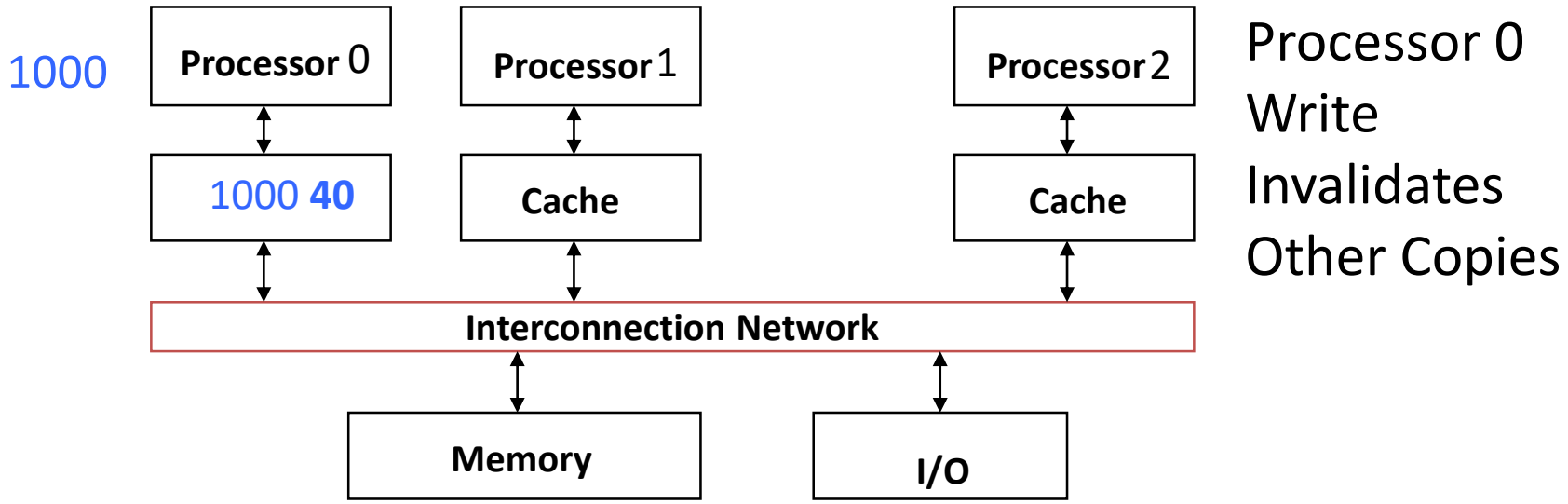
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



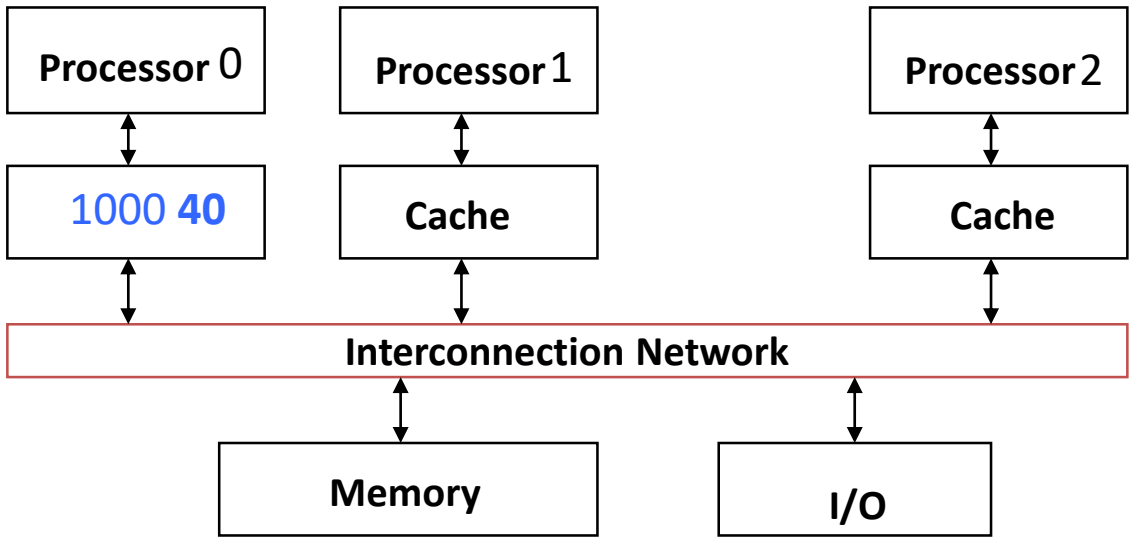
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



Shared Memory and Caches

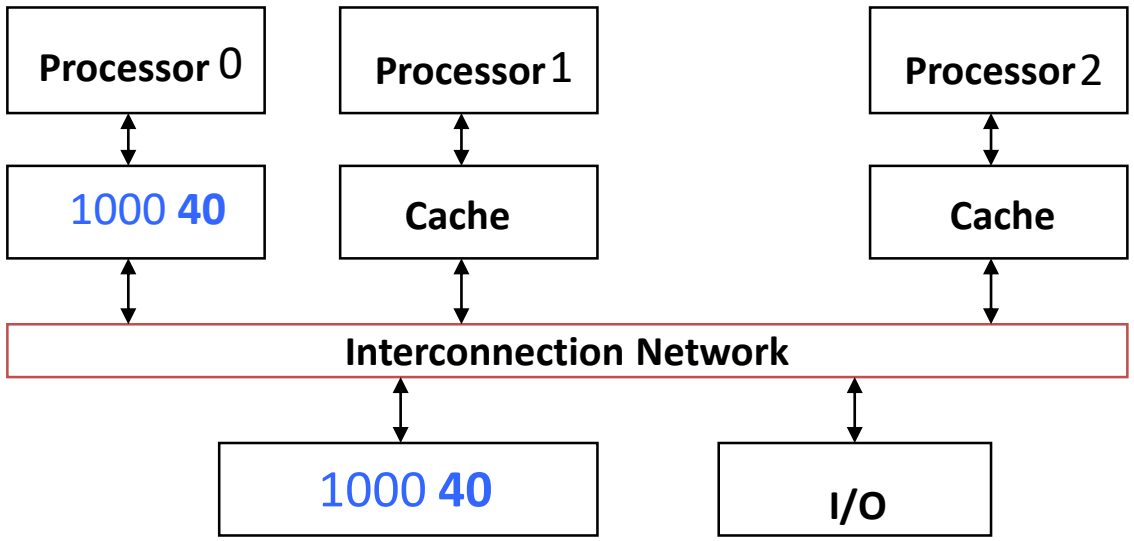
- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



Processor 0 Write Invalidates Other Copies

Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



Processor 0 Write Invalidates Other Copies

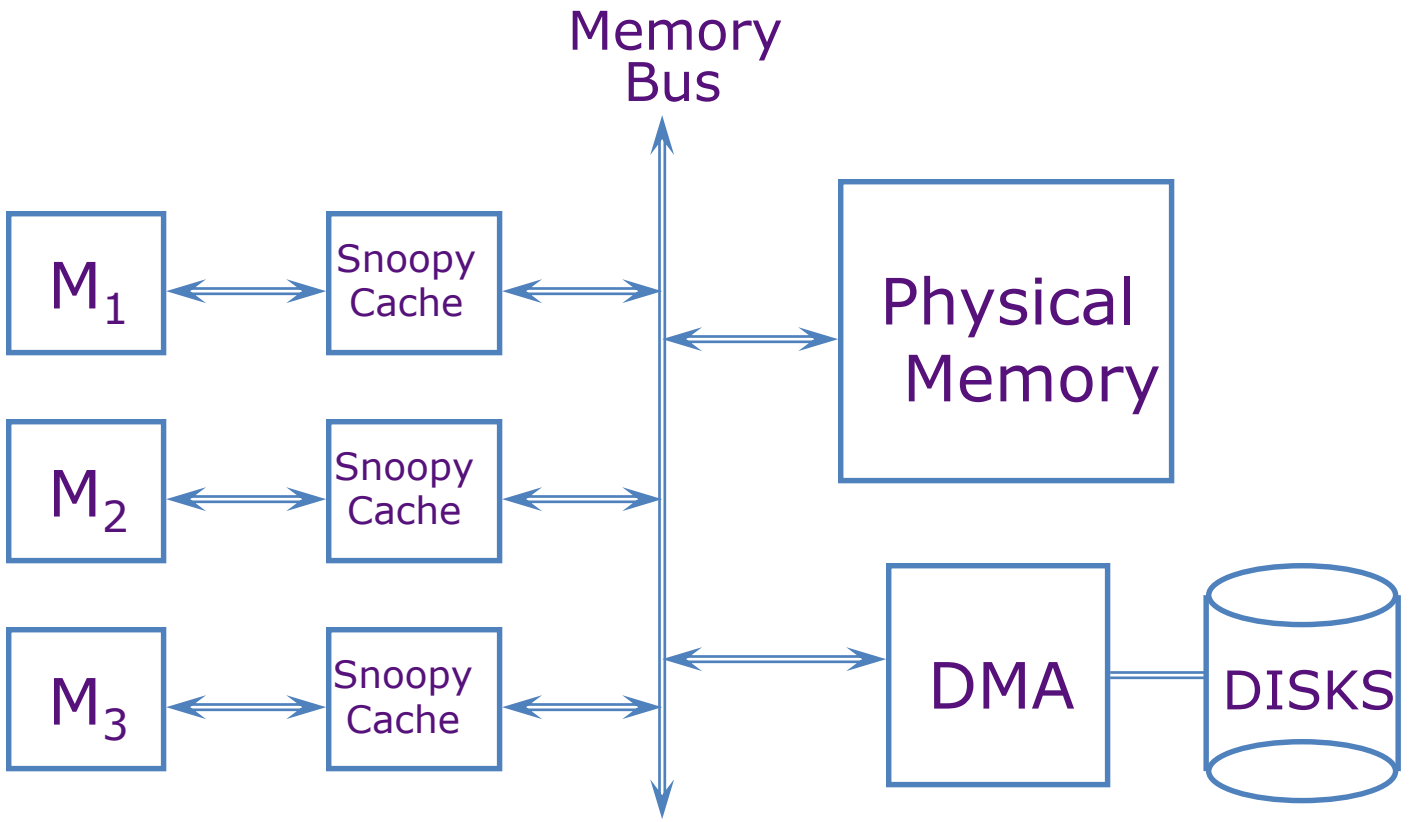
Keeping Multiple Caches Coherent

- **Architect's job**: shared memory => keep cache values coherent

- **Idea**: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate all other copies

- Shared written result can "ping-pong" between caches

Shared Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

Snoopy Cache Coherence Protocols

write miss:

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

read miss:

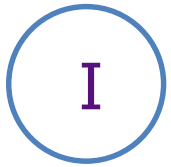
if a dirty copy is found in some cache, a write-back is performed before the memory is read

Cache State Transition Diagram

The MSI protocol

Each cache line has state bits

M: Modified
S: Shared
I: Invalid



Cache state in processor P_1

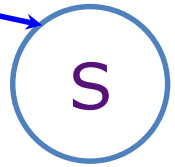
Cache State Transition Diagram

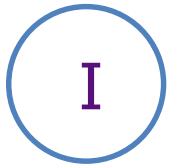
The MSI protocol

Each cache line has state bits

M: Modified
S: Shared
I: Invalid

Read miss
(P1 gets line from memory)





Cache state in processor P_1

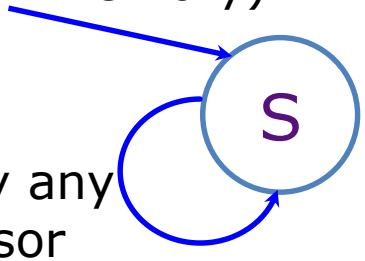
Cache State Transition Diagram

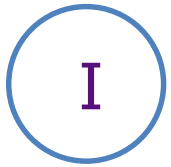
The MSI protocol

Each cache line has state bits

M: Modified
S: Shared
I: Invalid

Read miss
(P1 gets line from memory)





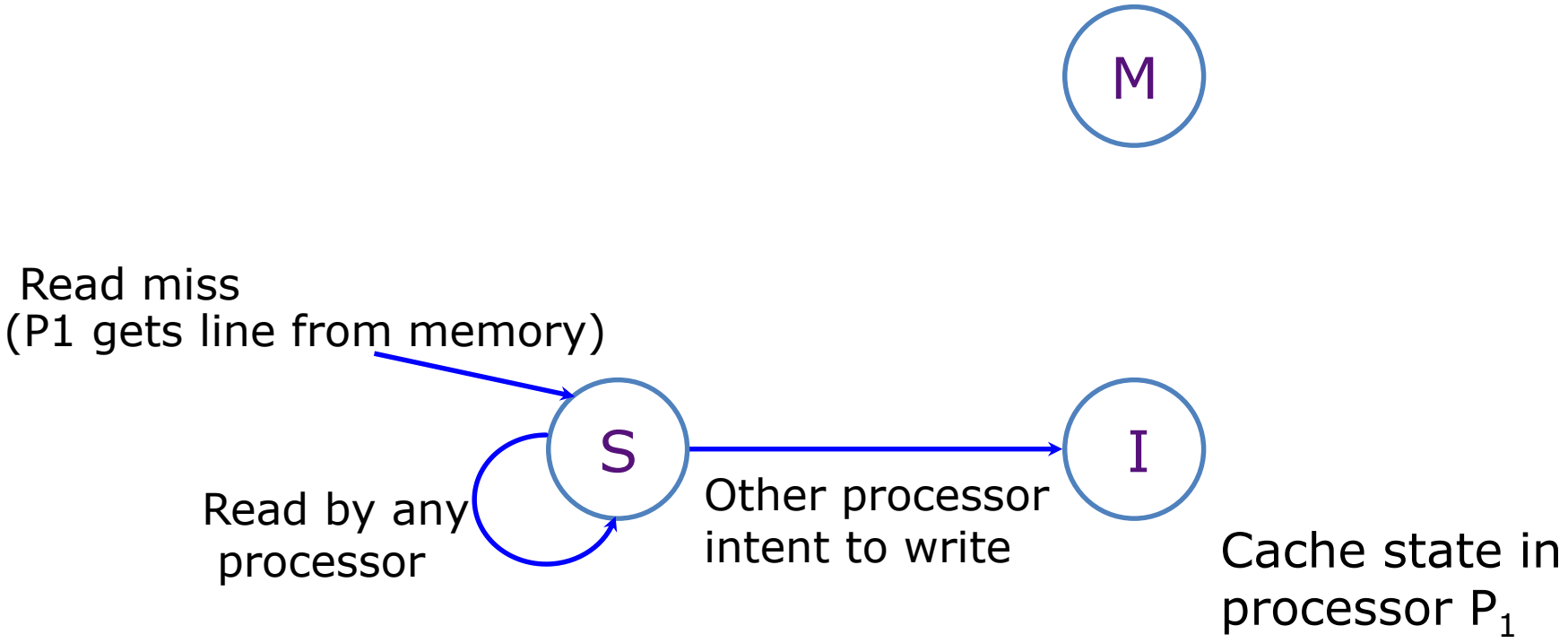
Cache state in processor P_1

Cache State Transition Diagram

The MSI protocol

Each cache line has state bits

M: Modified
S: Shared
I: Invalid

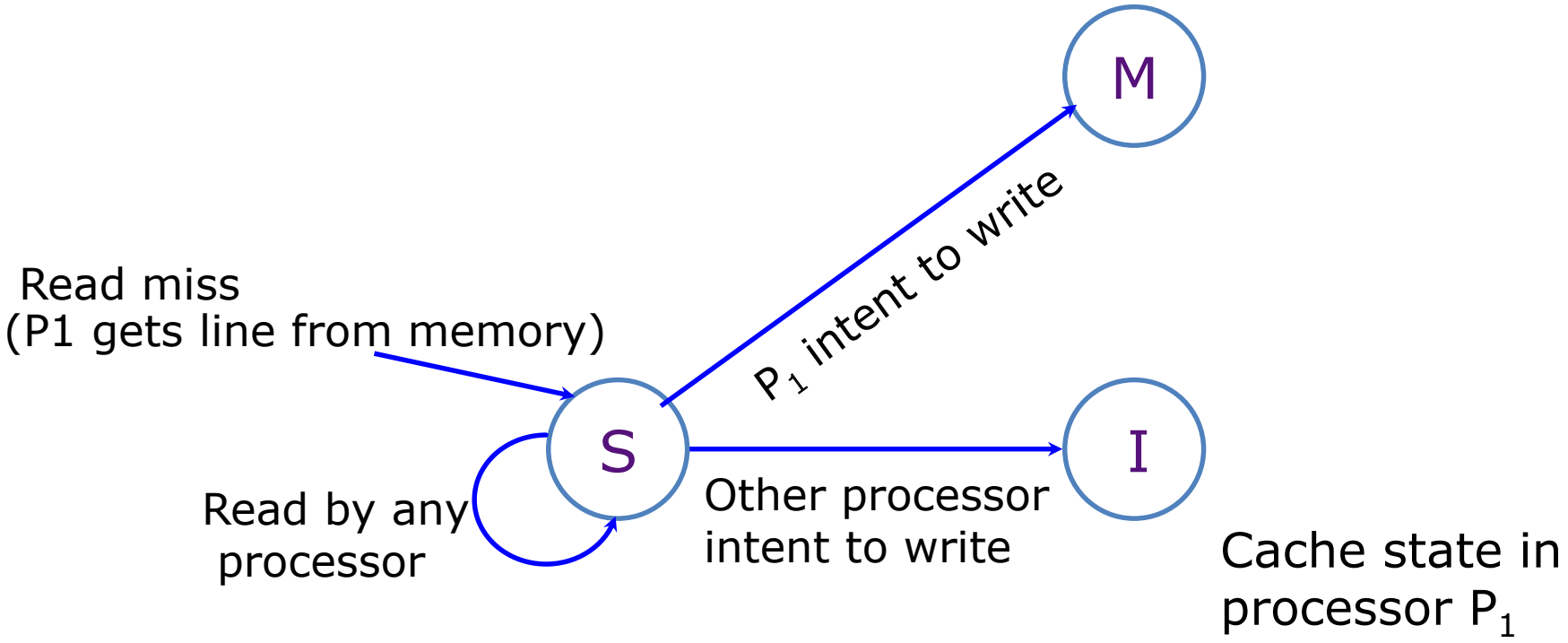


Cache State Transition Diagram

The MSI protocol

Each cache line has state bits

M: Modified
S: Shared
I: Invalid

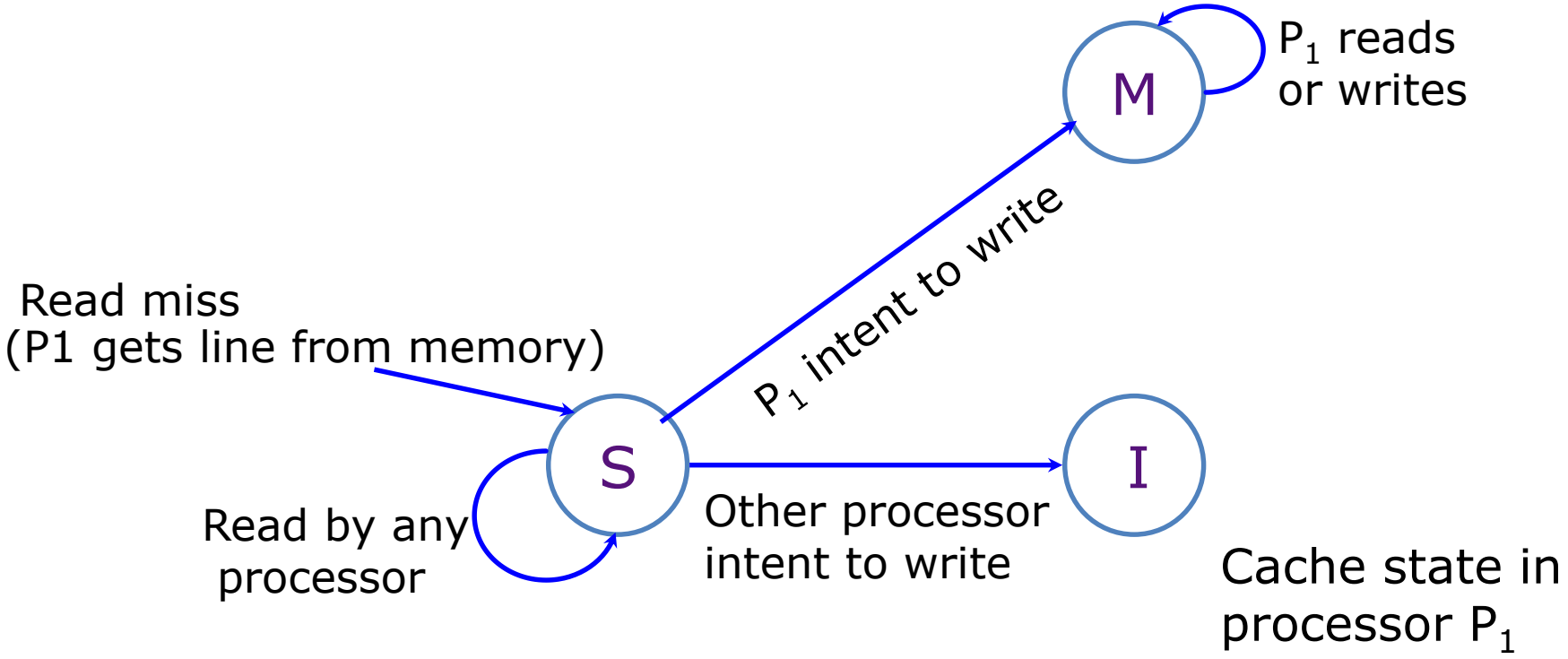


Cache State Transition Diagram

The MSI protocol

Each cache line has state bits

M: Modified
S: Shared
I: Invalid

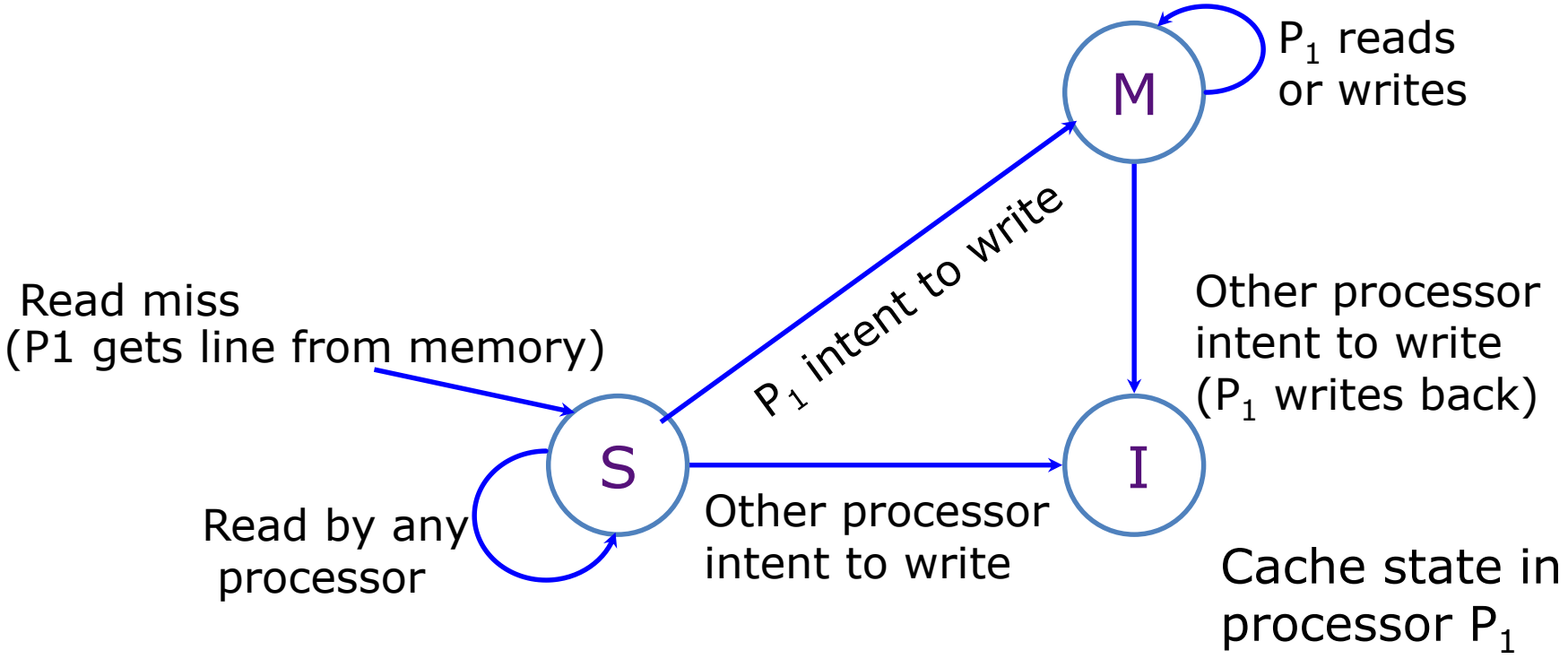


Cache State Transition Diagram

The MSI protocol

Each cache line has state bits

M: Modified
S: Shared
I: Invalid

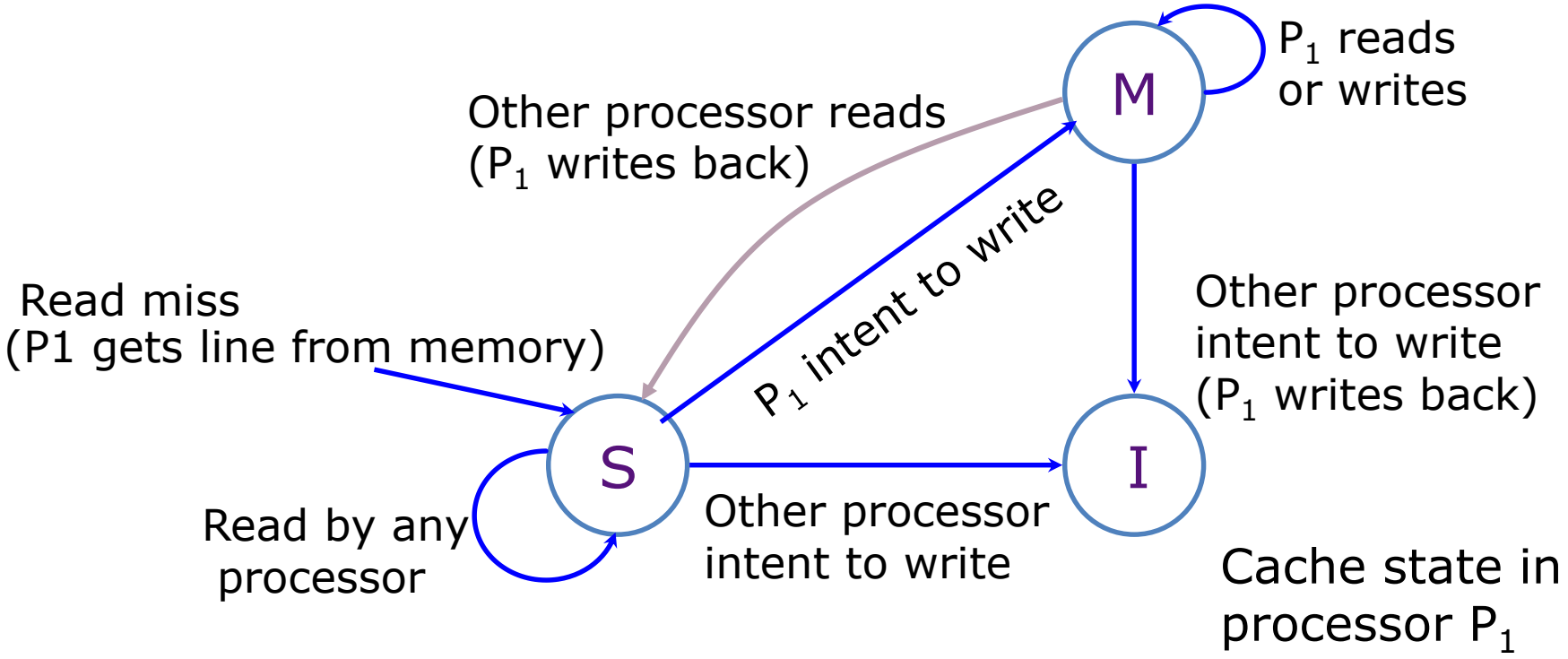


Cache State Transition Diagram

The MSI protocol

Each cache line has state bits

M: Modified
S: Shared
I: Invalid

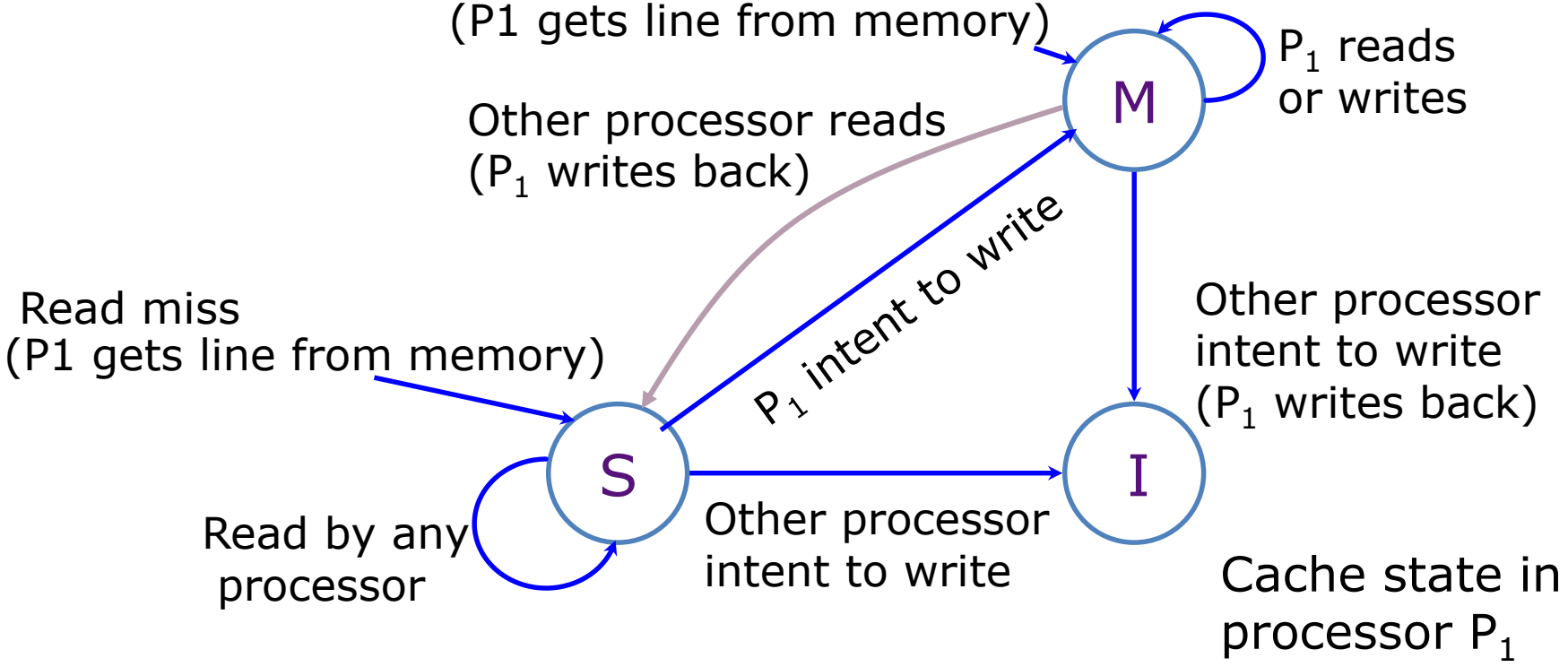


Cache State Transition Diagram

The MSI protocol

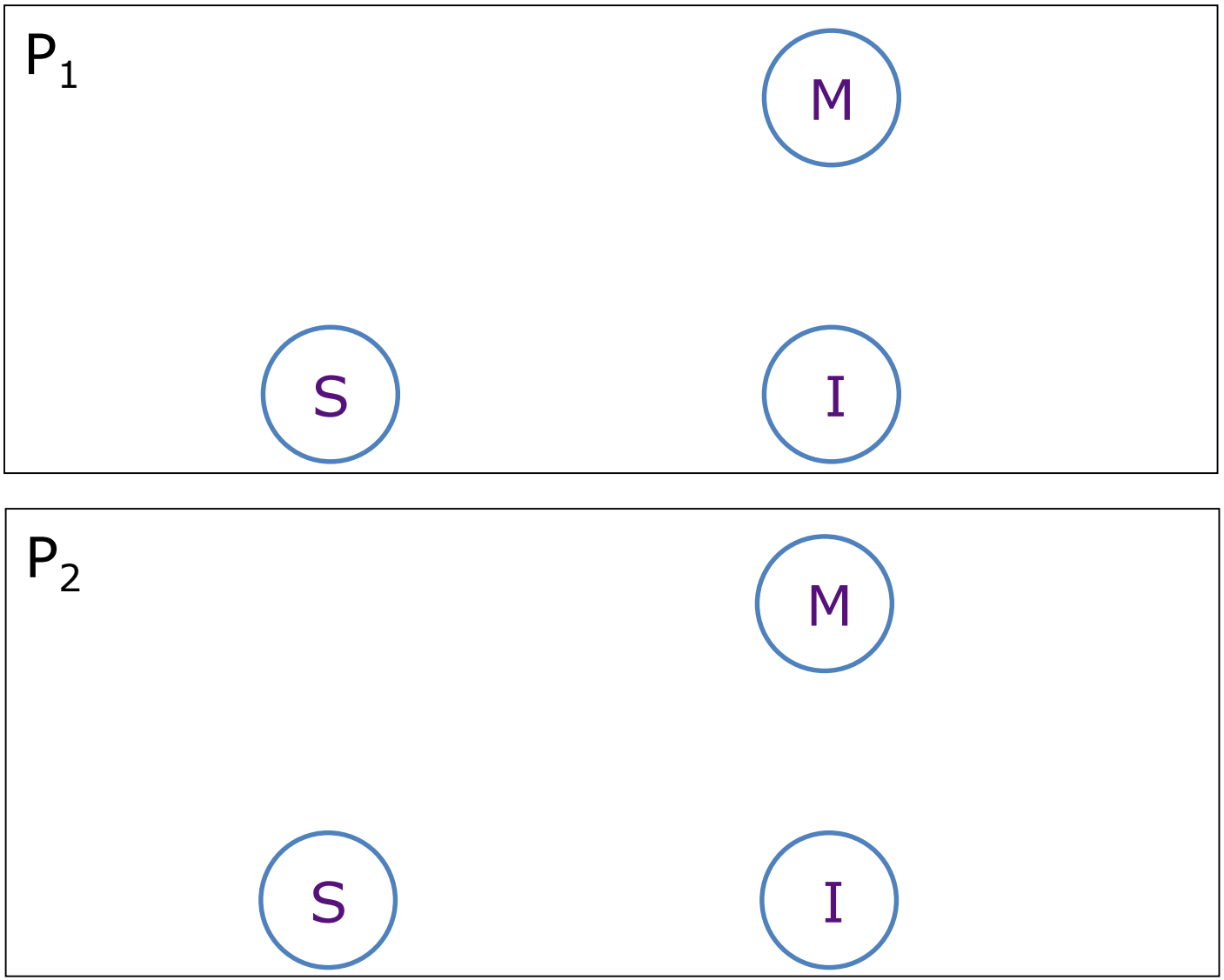
Each cache line has state bits

M: Modified
S: Shared
I: Invalid



Two Processor Example

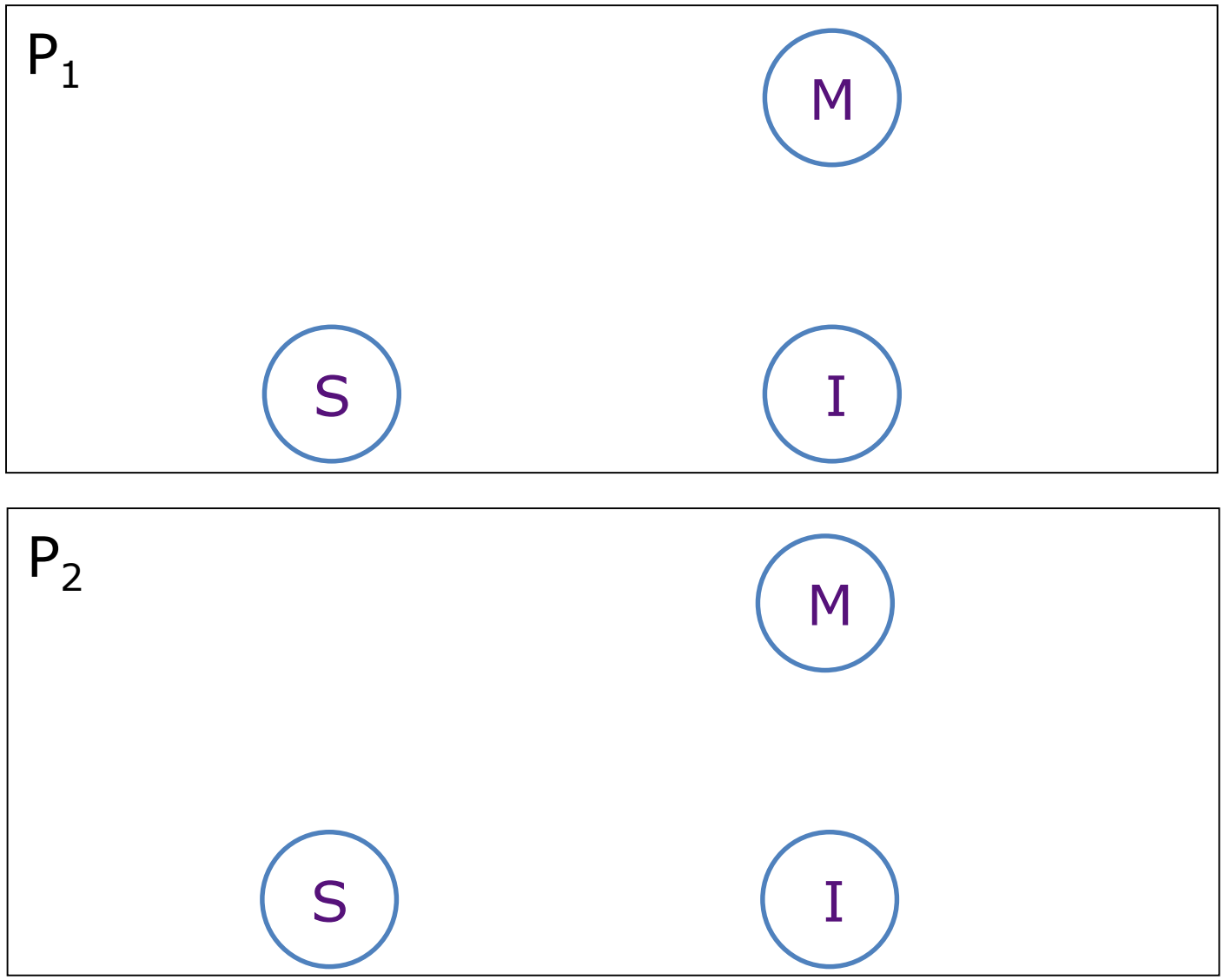
(Reading and writing the same cache line)



Two Processor Example

(Reading and writing the same cache line)

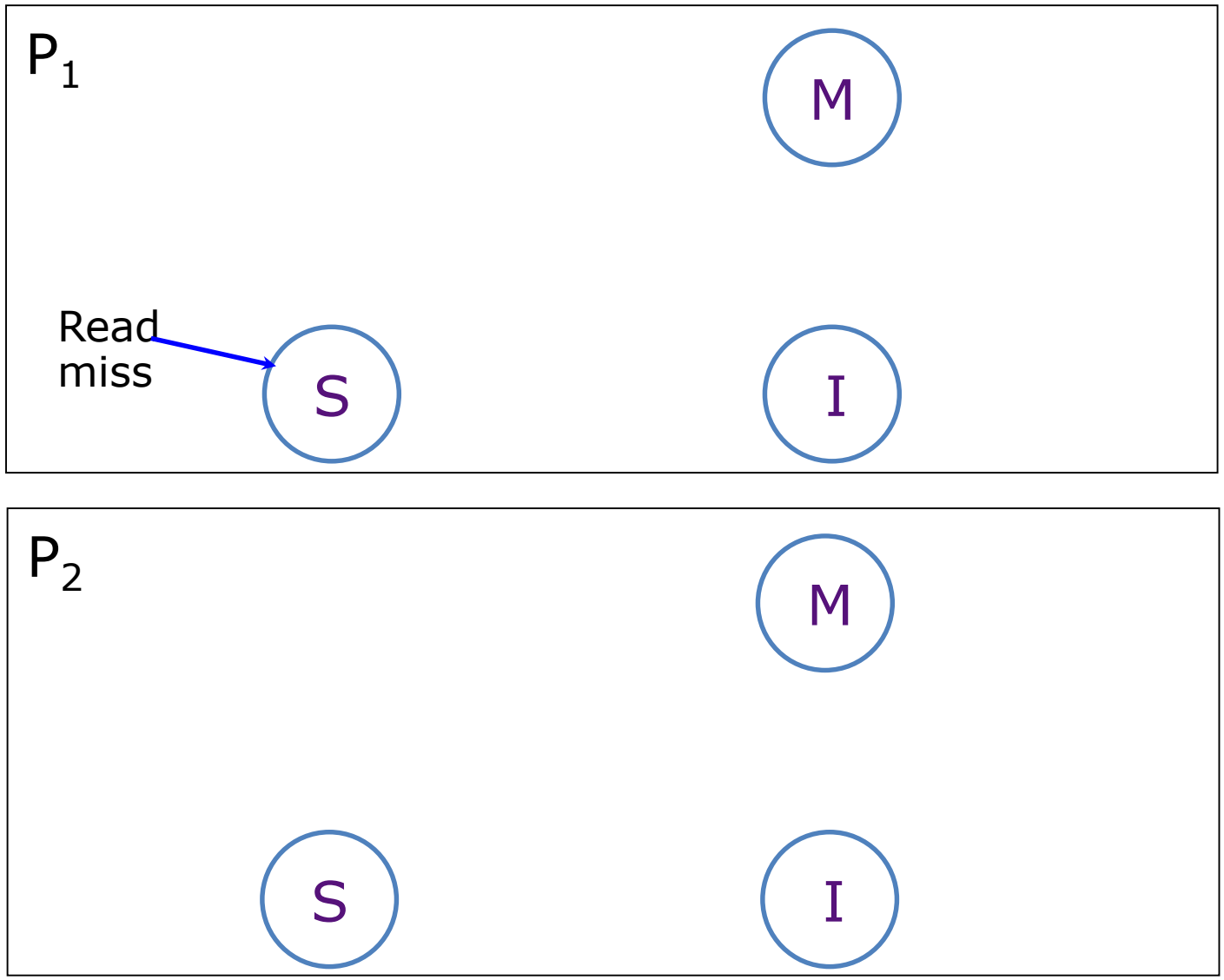
P_1 reads



Two Processor Example

(Reading and writing the same cache line)

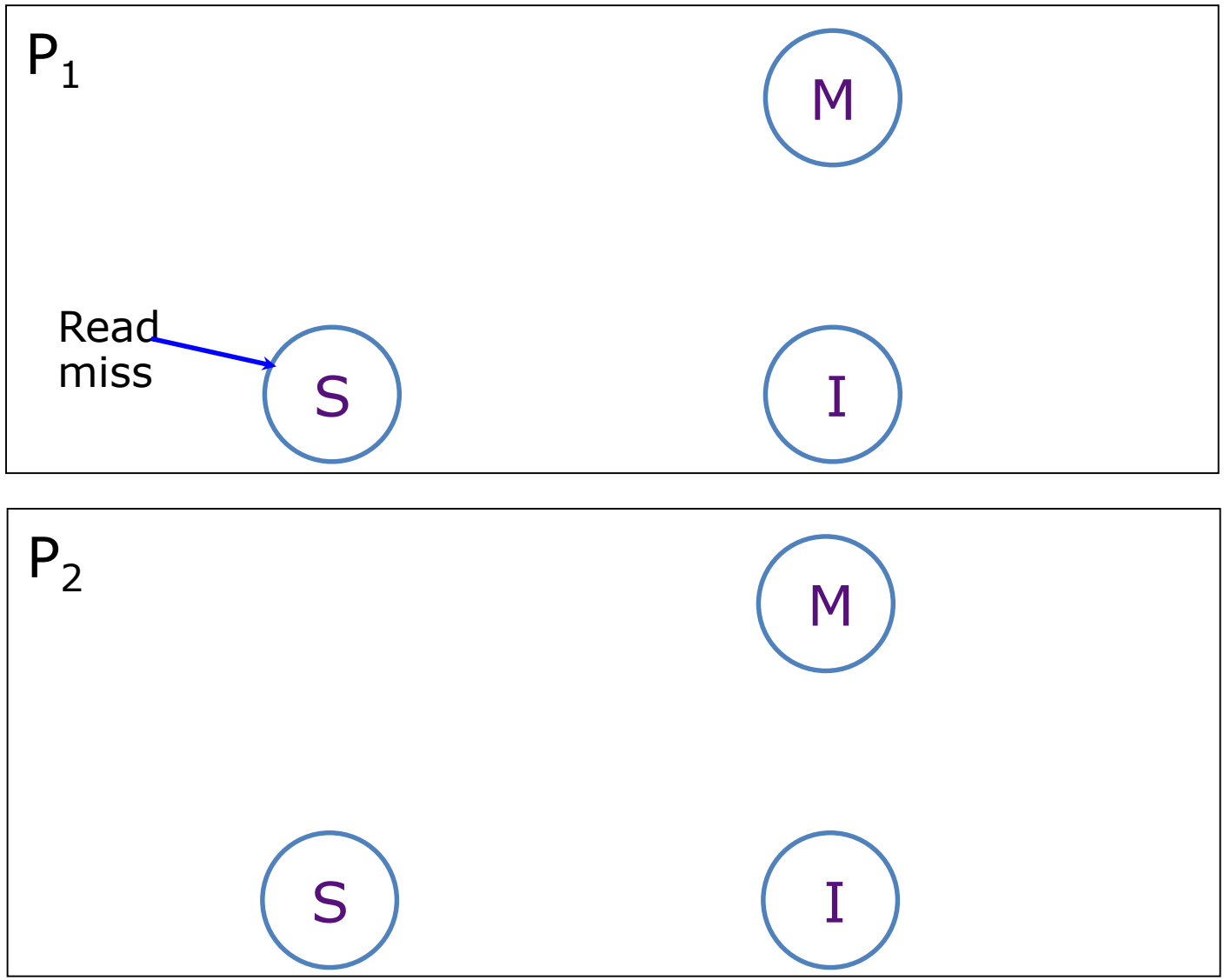
P_1 reads



Two Processor Example

(Reading and writing the same cache line)

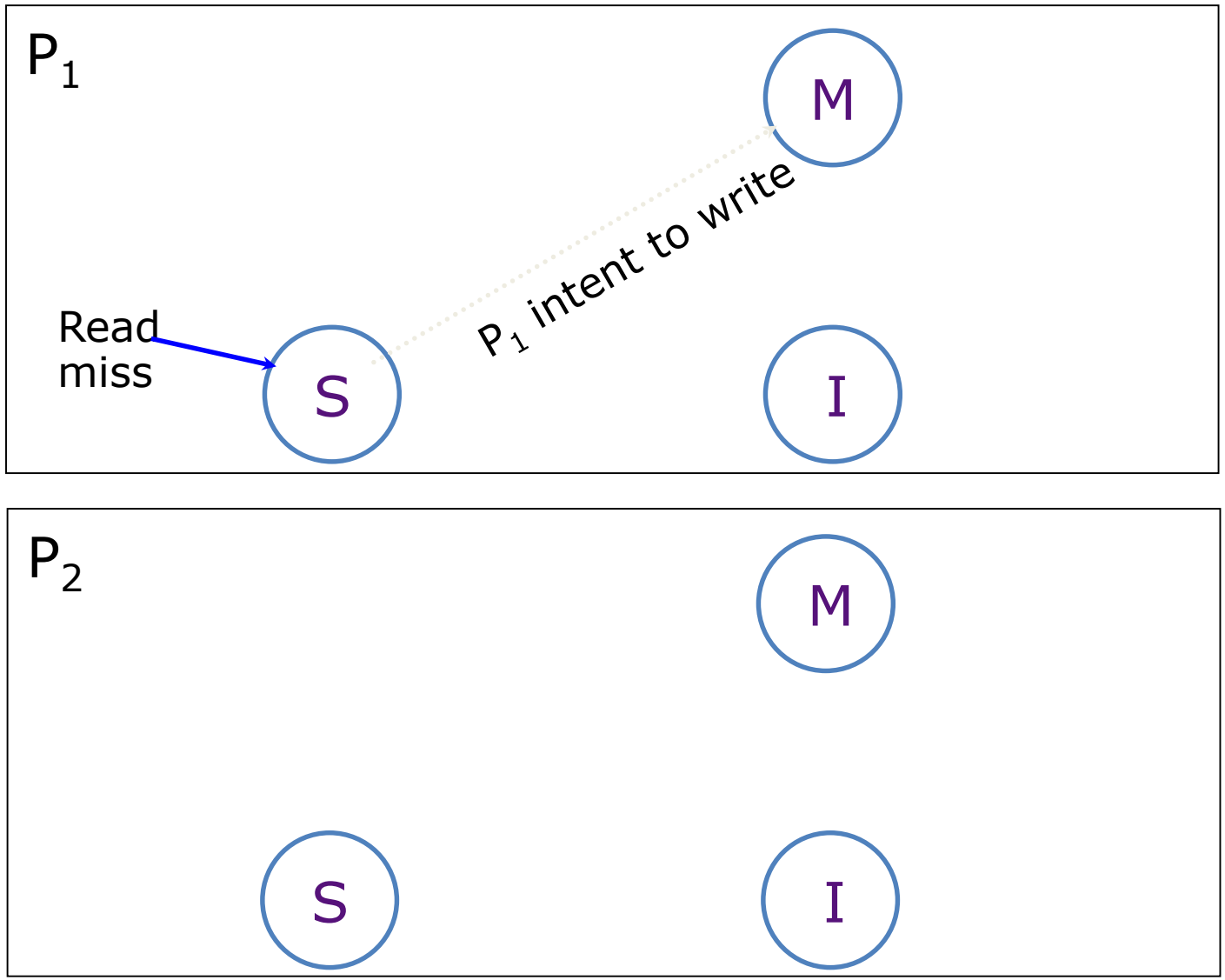
P_1 reads
P_1 writes



Two Processor Example

(Reading and writing the same cache line)

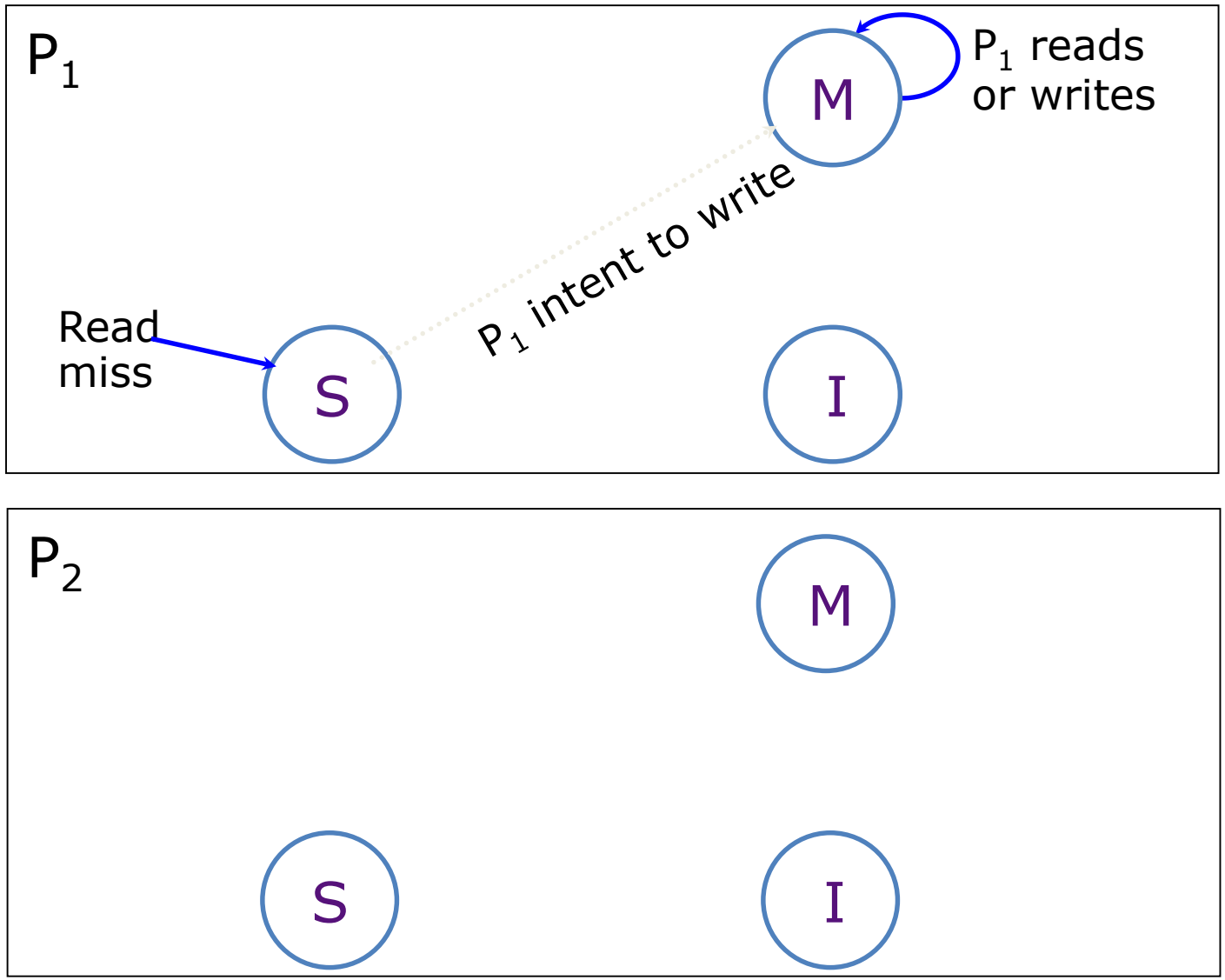
P_1 reads
P_1 writes



Two Processor Example

(Reading and writing the same cache line)

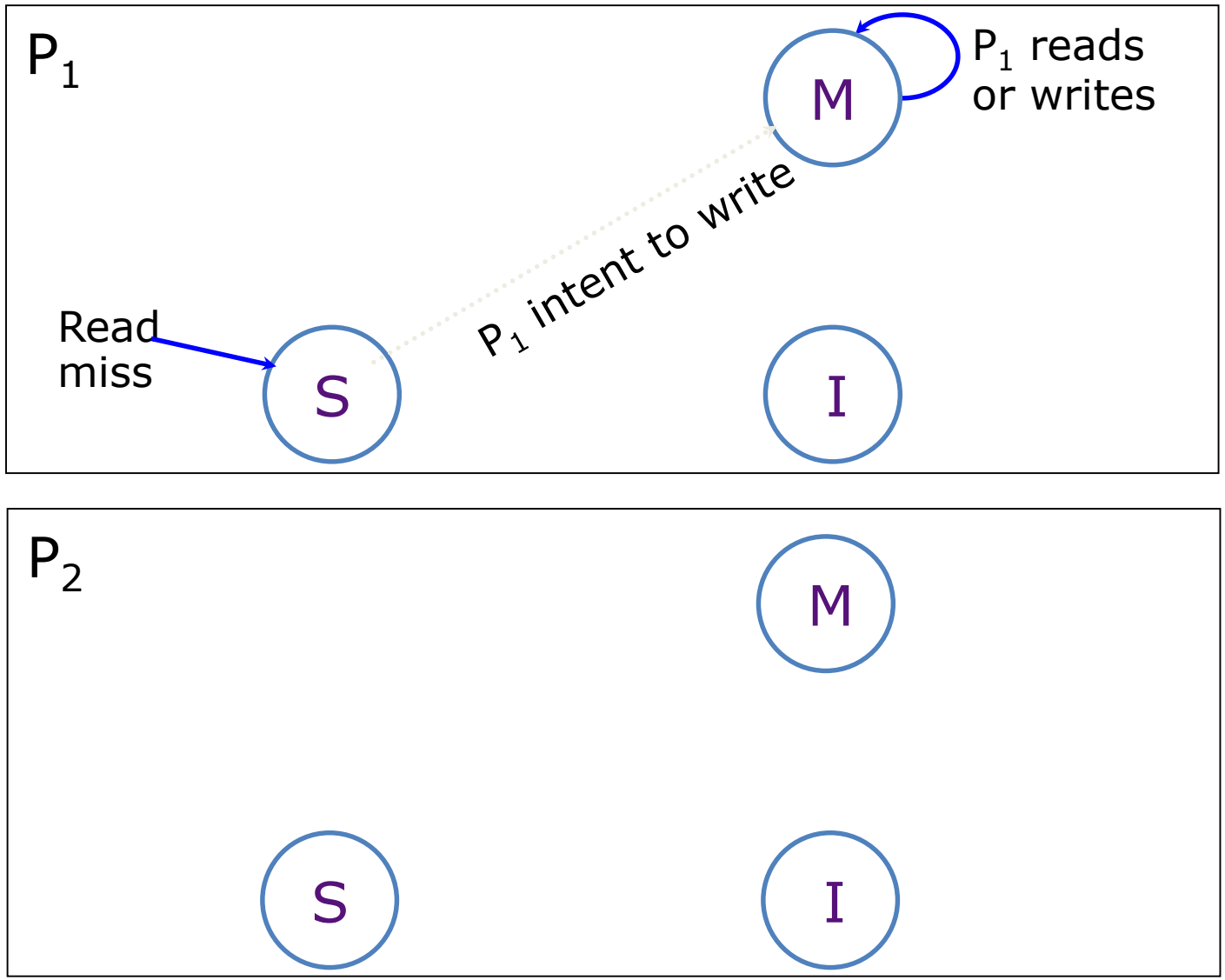
P_1 reads
P_1 writes



Two Processor Example

(Reading and writing the same cache line)

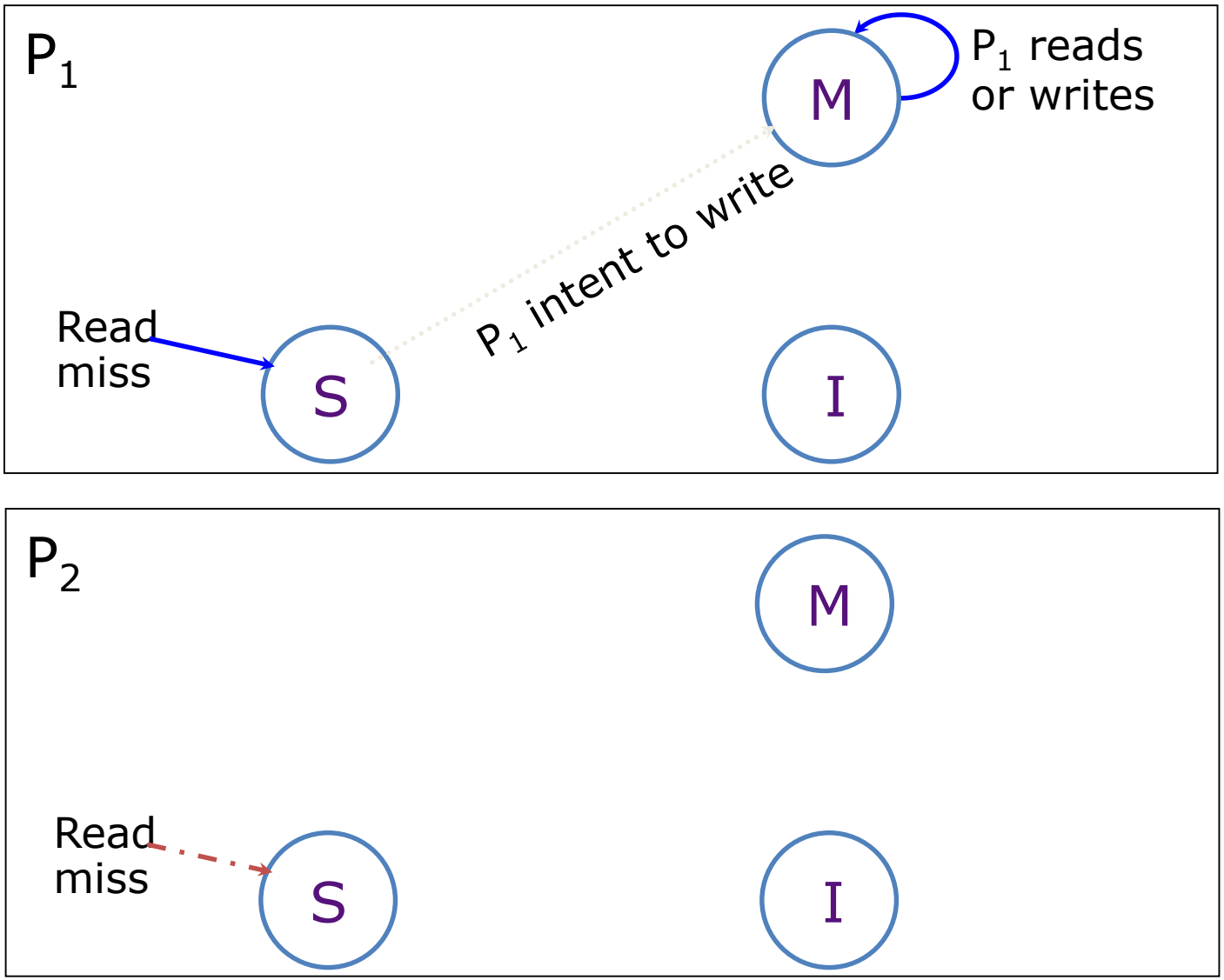
P_1 reads
P_1 writes
P_2 reads



Two Processor Example

(Reading and writing the same cache line)

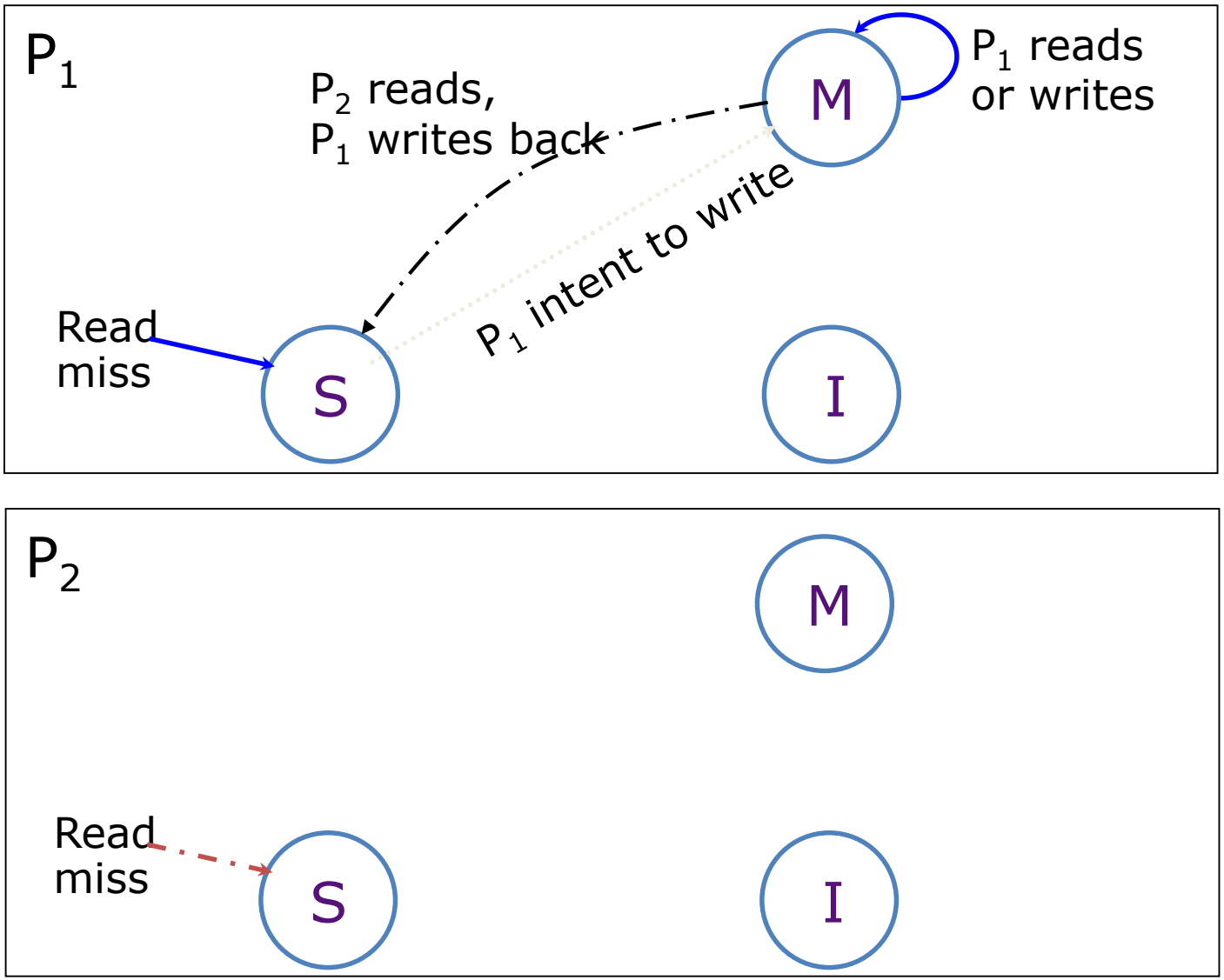
P_1 reads
P_1 writes
P_2 reads



Two Processor Example

(Reading and writing the same cache line)

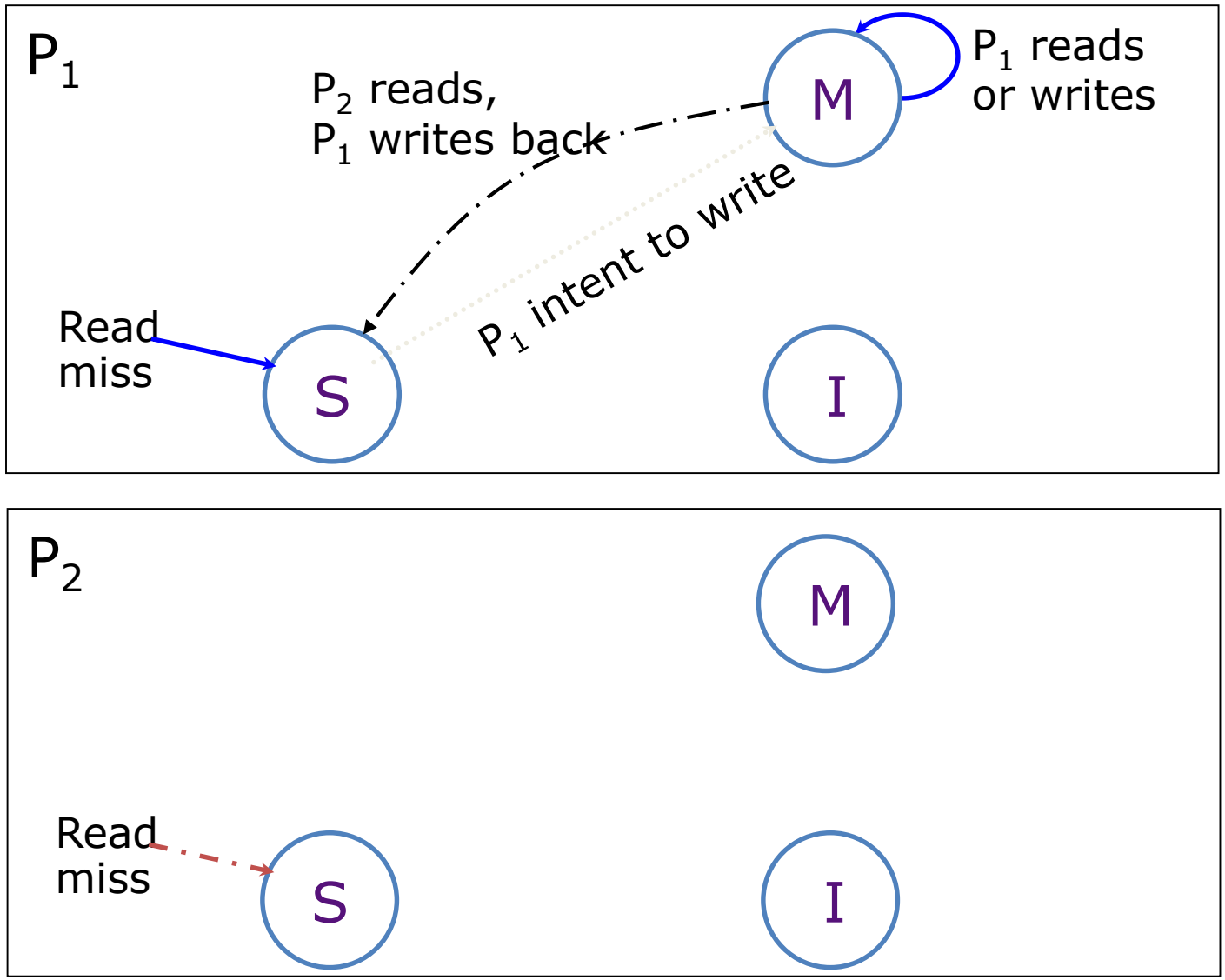
P_1 reads
P_1 writes
P_2 reads



Two Processor Example

(Reading and writing the same cache line)

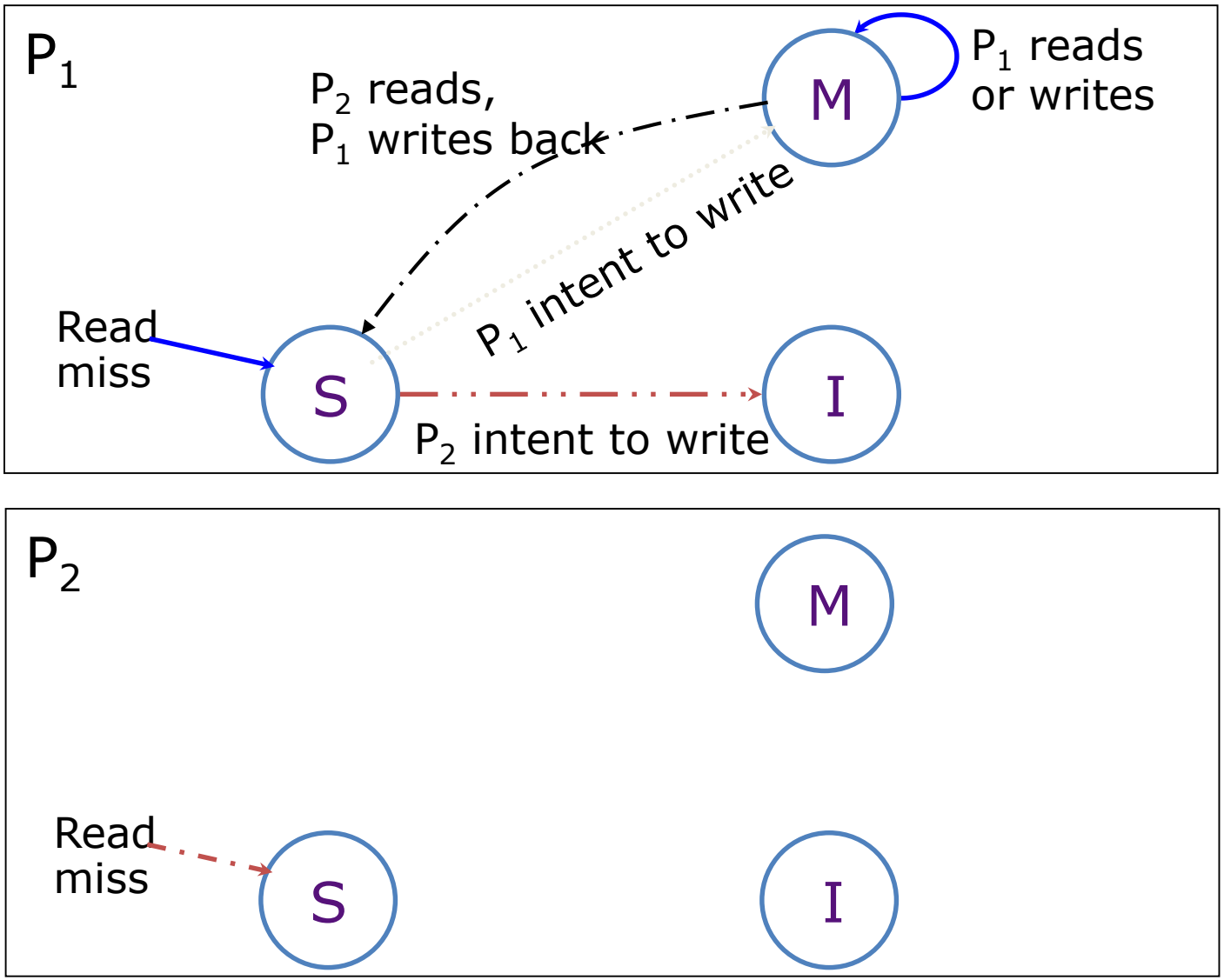
P_1 reads
P_1 writes
P_2 reads
P_2 writes



Two Processor Example

(Reading and writing the same cache line)

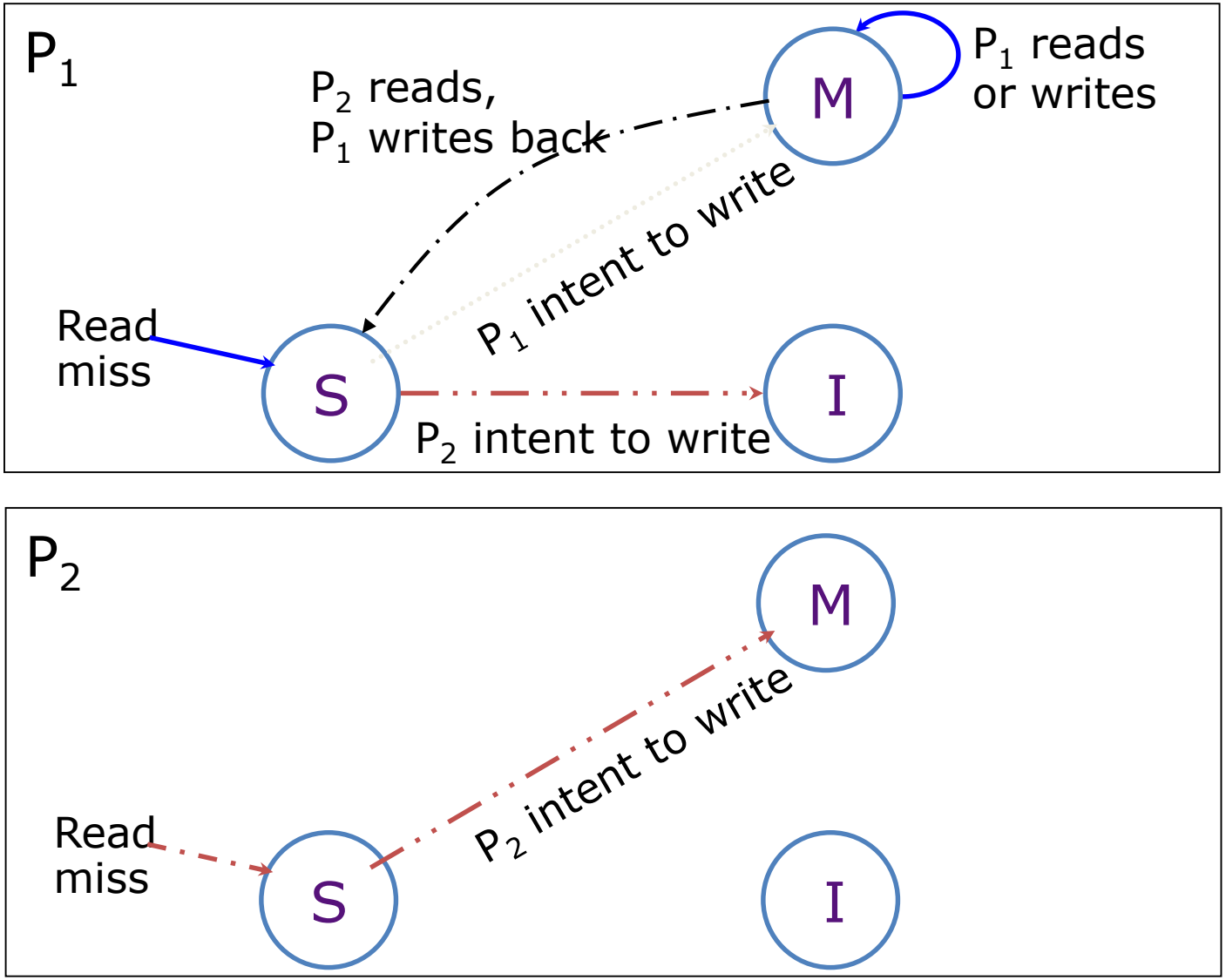
P_1 reads
P_1 writes
P_2 reads
P_2 writes



Two Processor Example

(Reading and writing the same cache line)

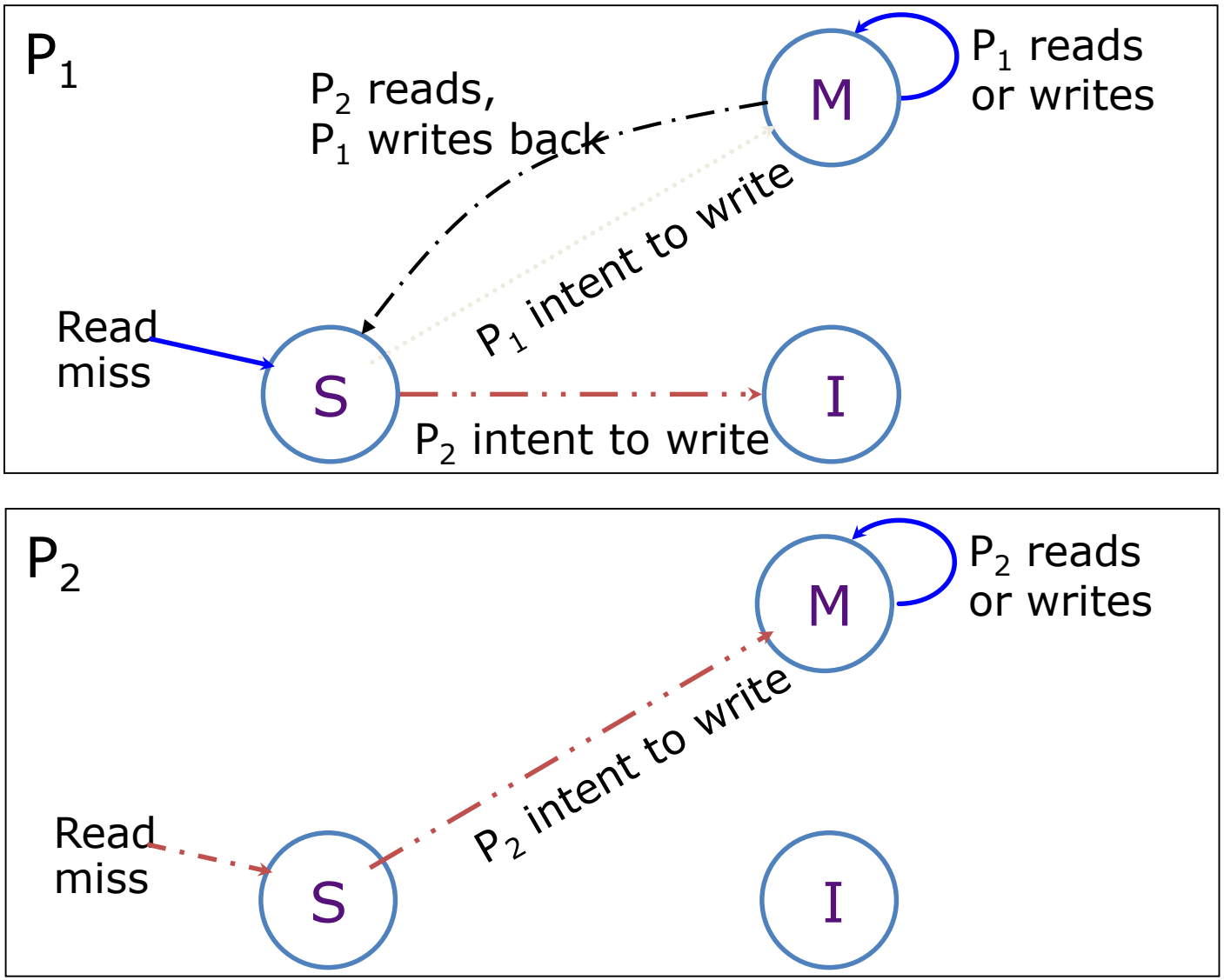
P_1 reads
P_1 writes
P_2 reads
P_2 writes



Two Processor Example

(Reading and writing the same cache line)

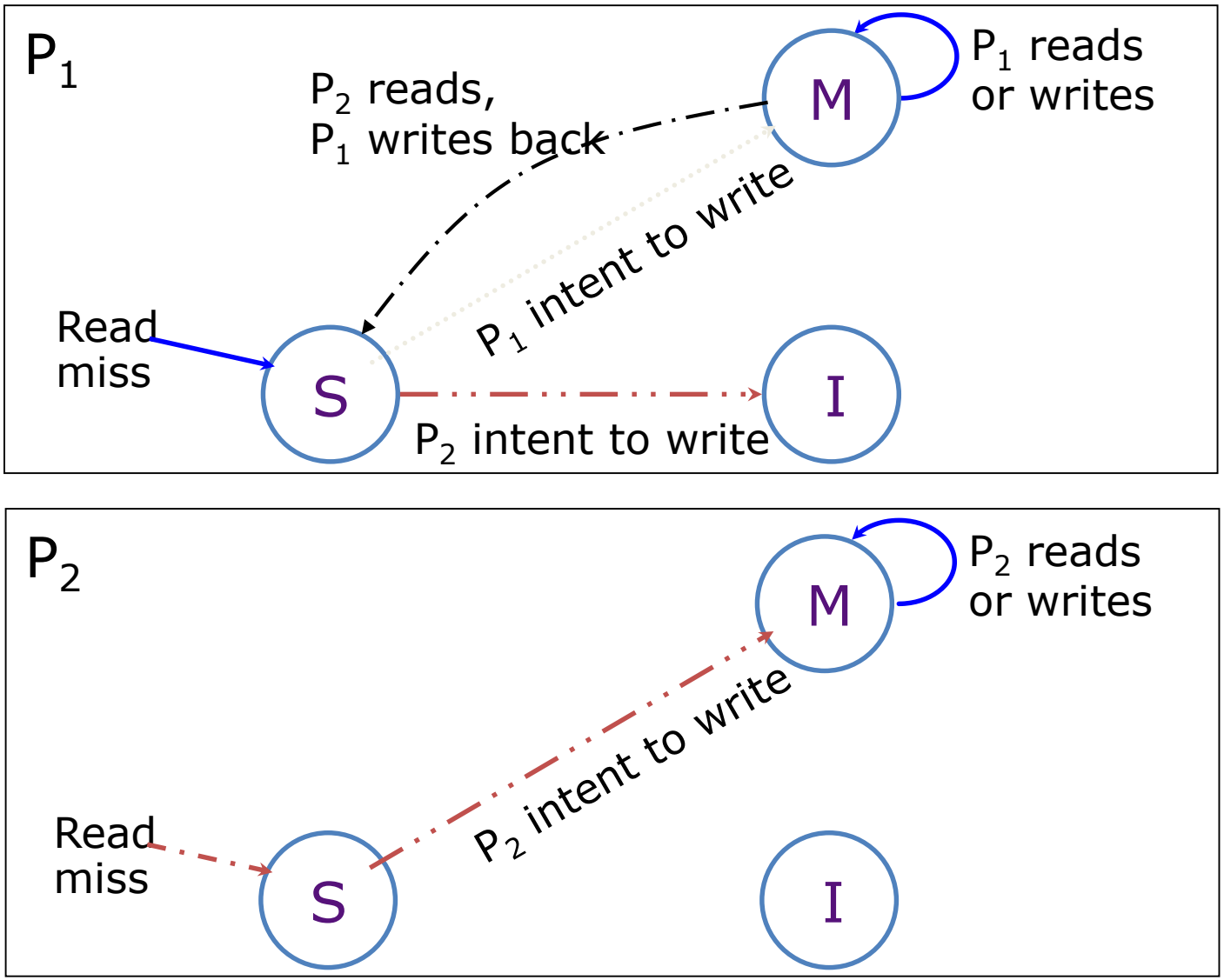
P_1 reads
P_1 writes
P_2 reads
P_2 writes



Two Processor Example

(Reading and writing the same cache line)

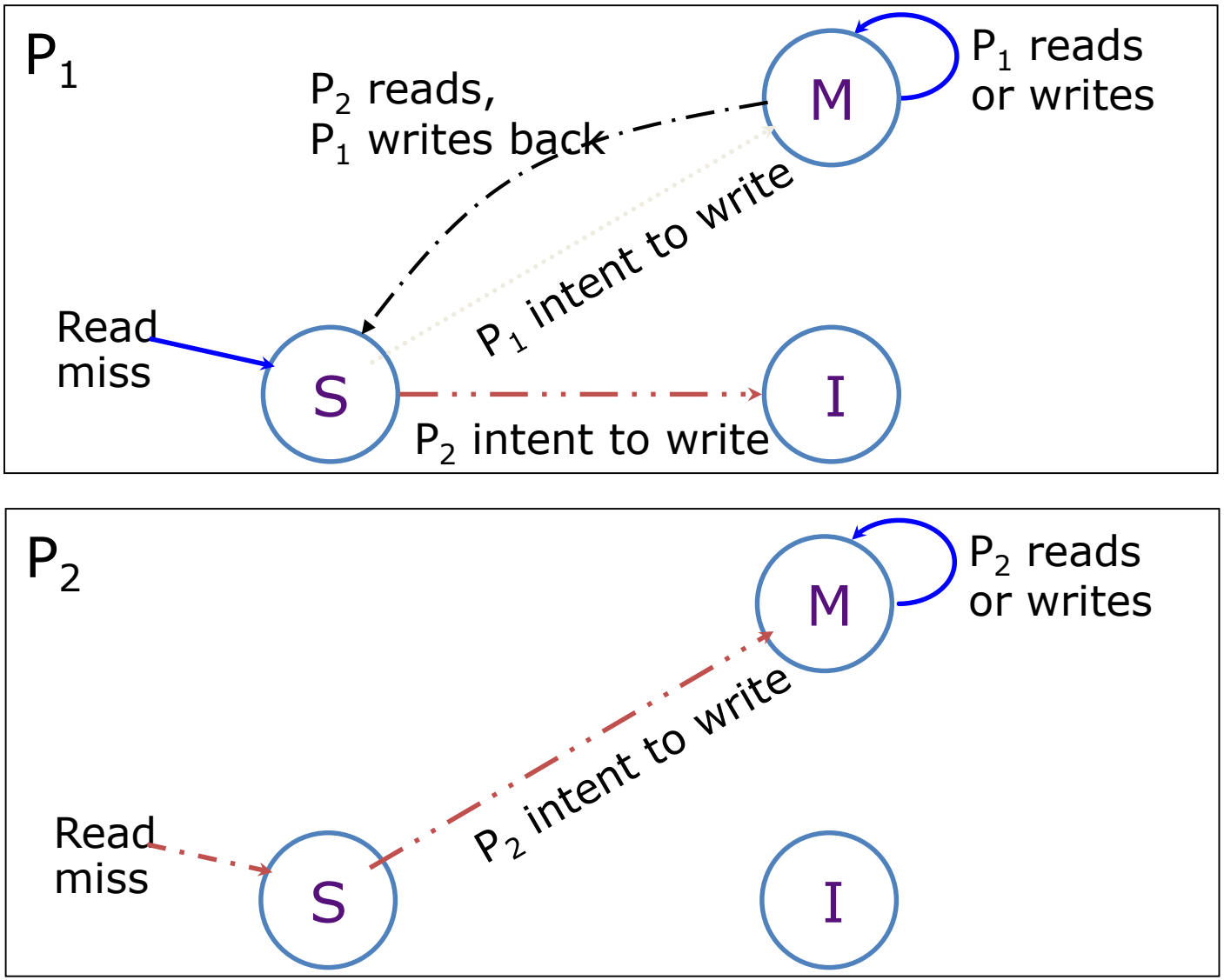
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads



Two Processor Example

(Reading and writing the same cache line)

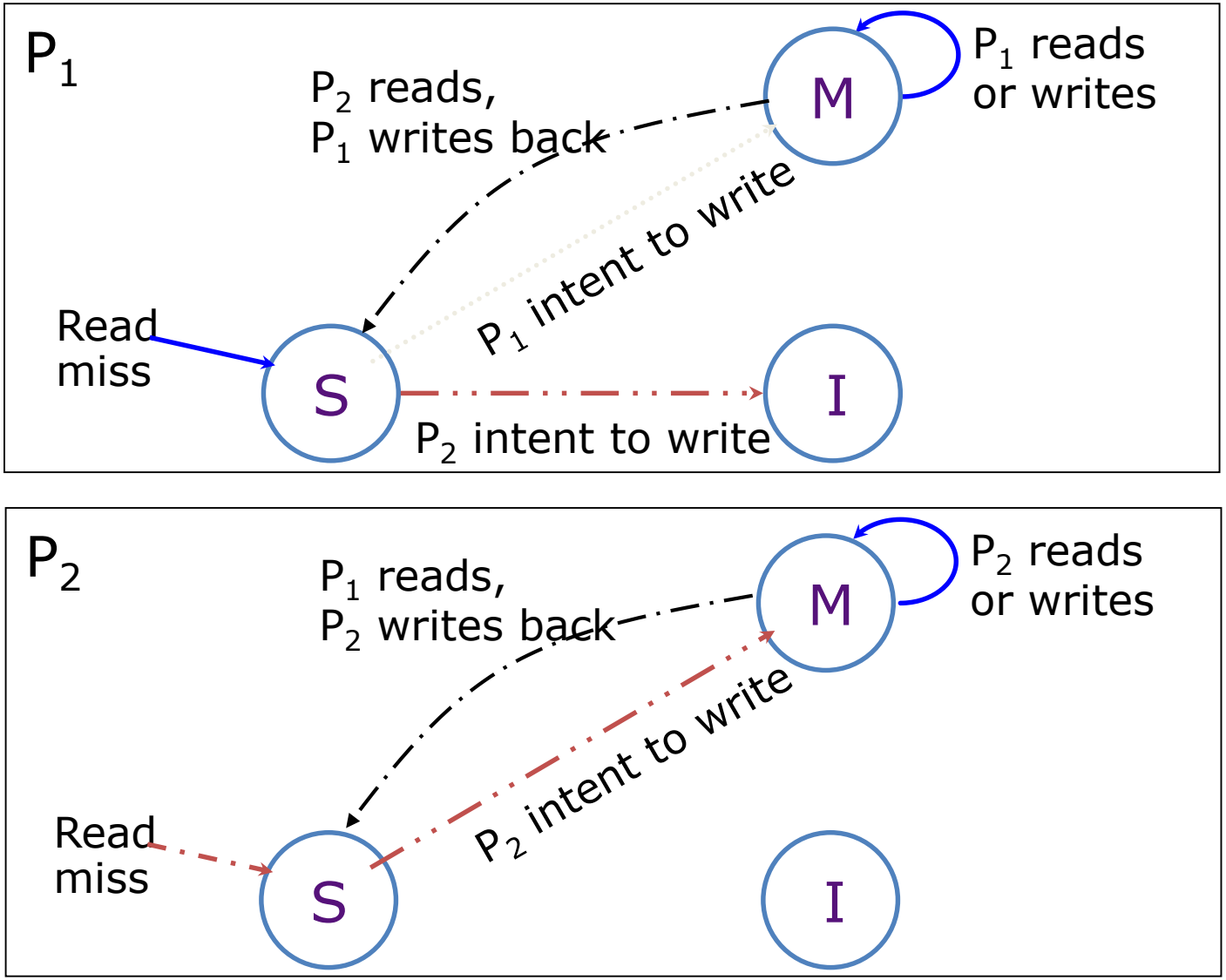
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads



Two Processor Example

(Reading and writing the same cache line)

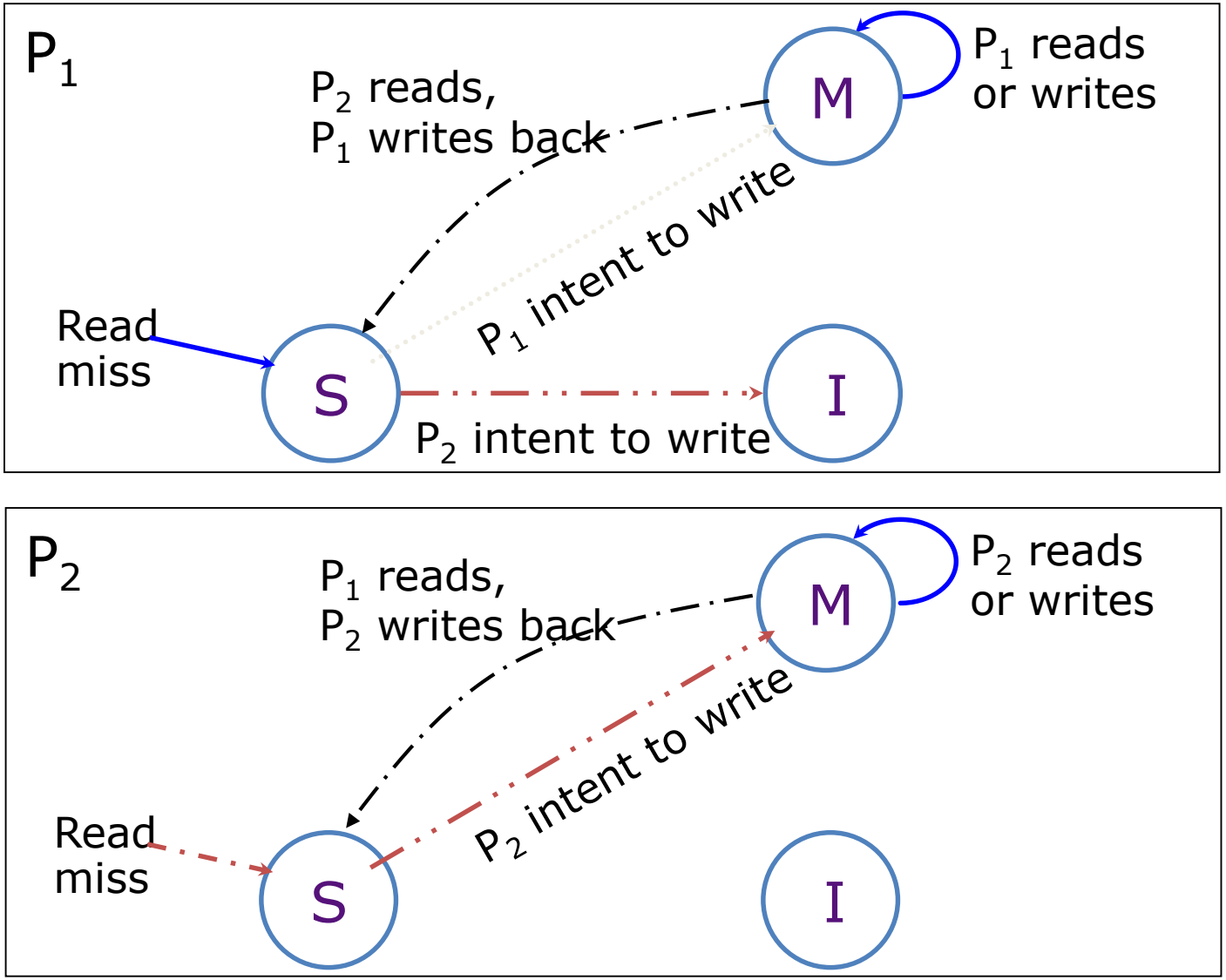
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads



Two Processor Example

(Reading and writing the same cache line)

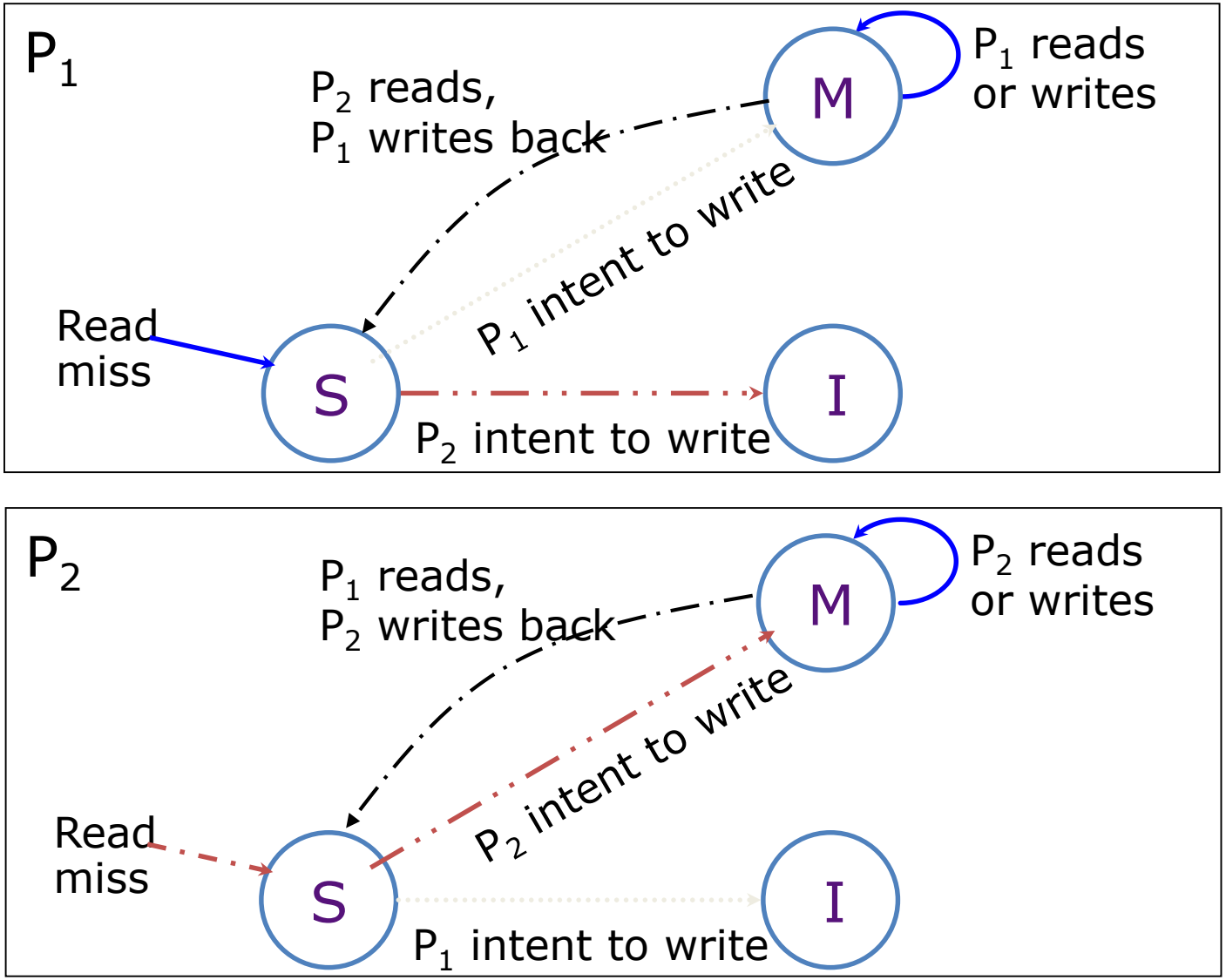
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes



Two Processor Example

(Reading and writing the same cache line)

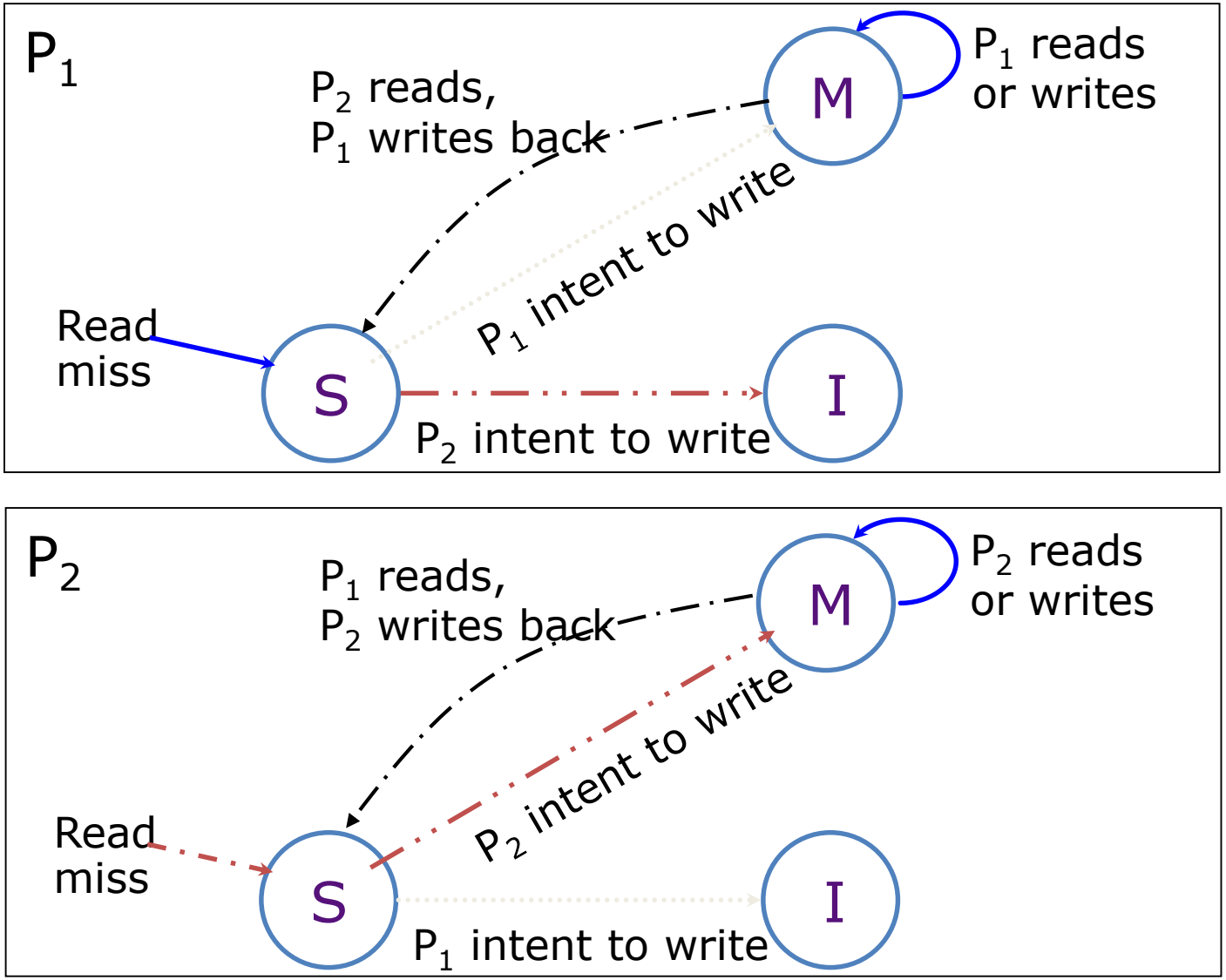
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes



Two Processor Example

(Reading and writing the same cache line)

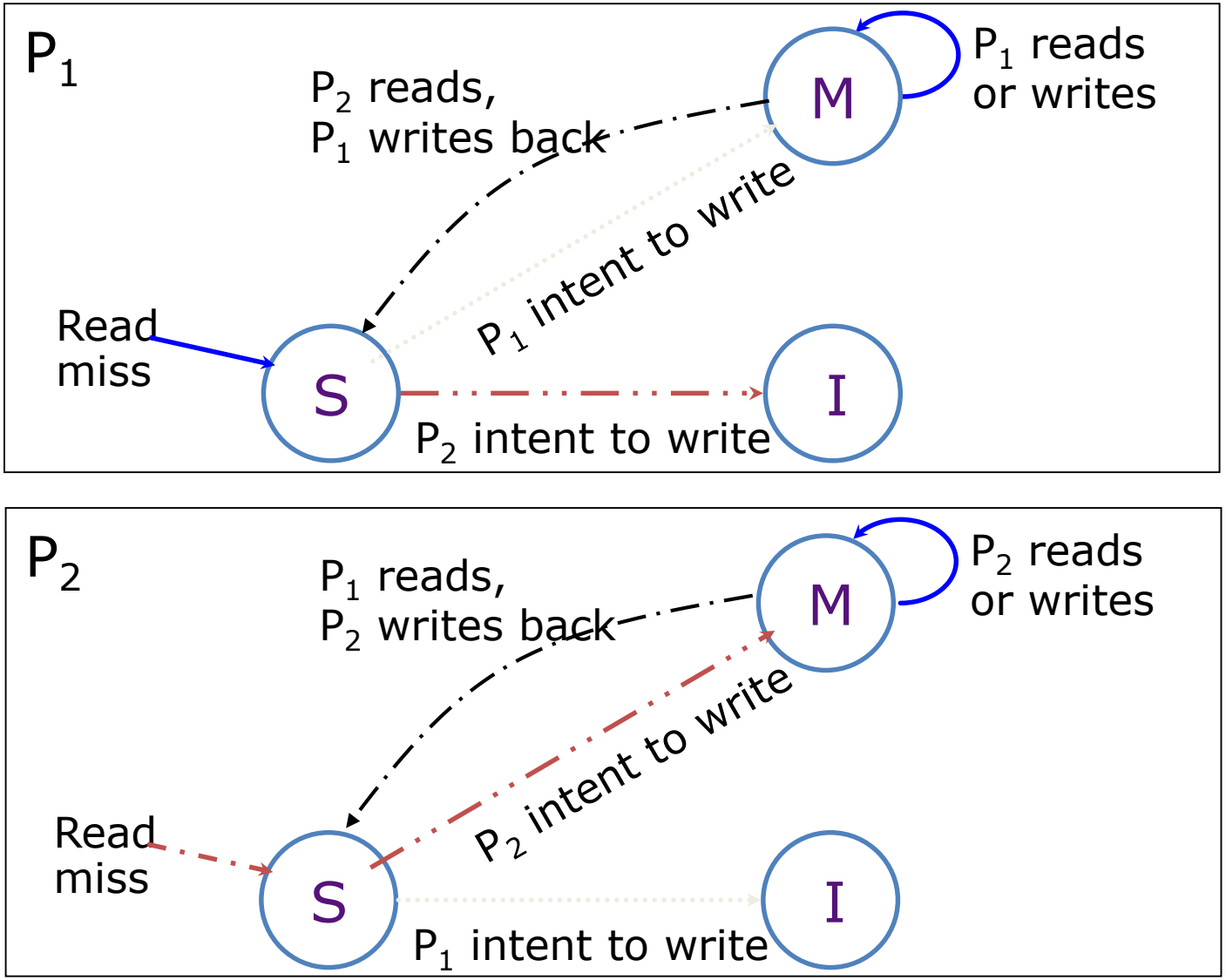
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes



Two Processor Example

(Reading and writing the same cache line)

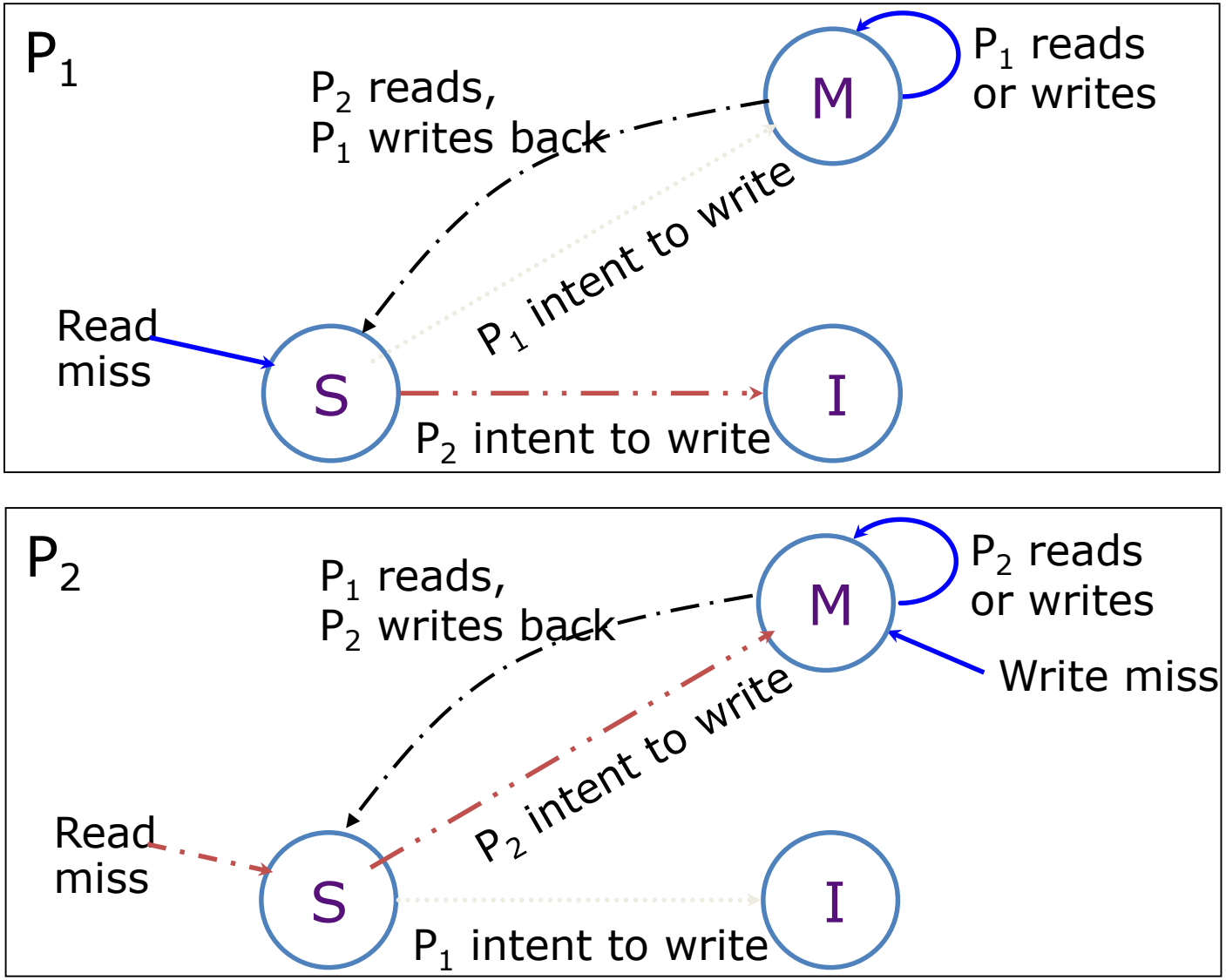
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes



Two Processor Example

(Reading and writing the same cache line)

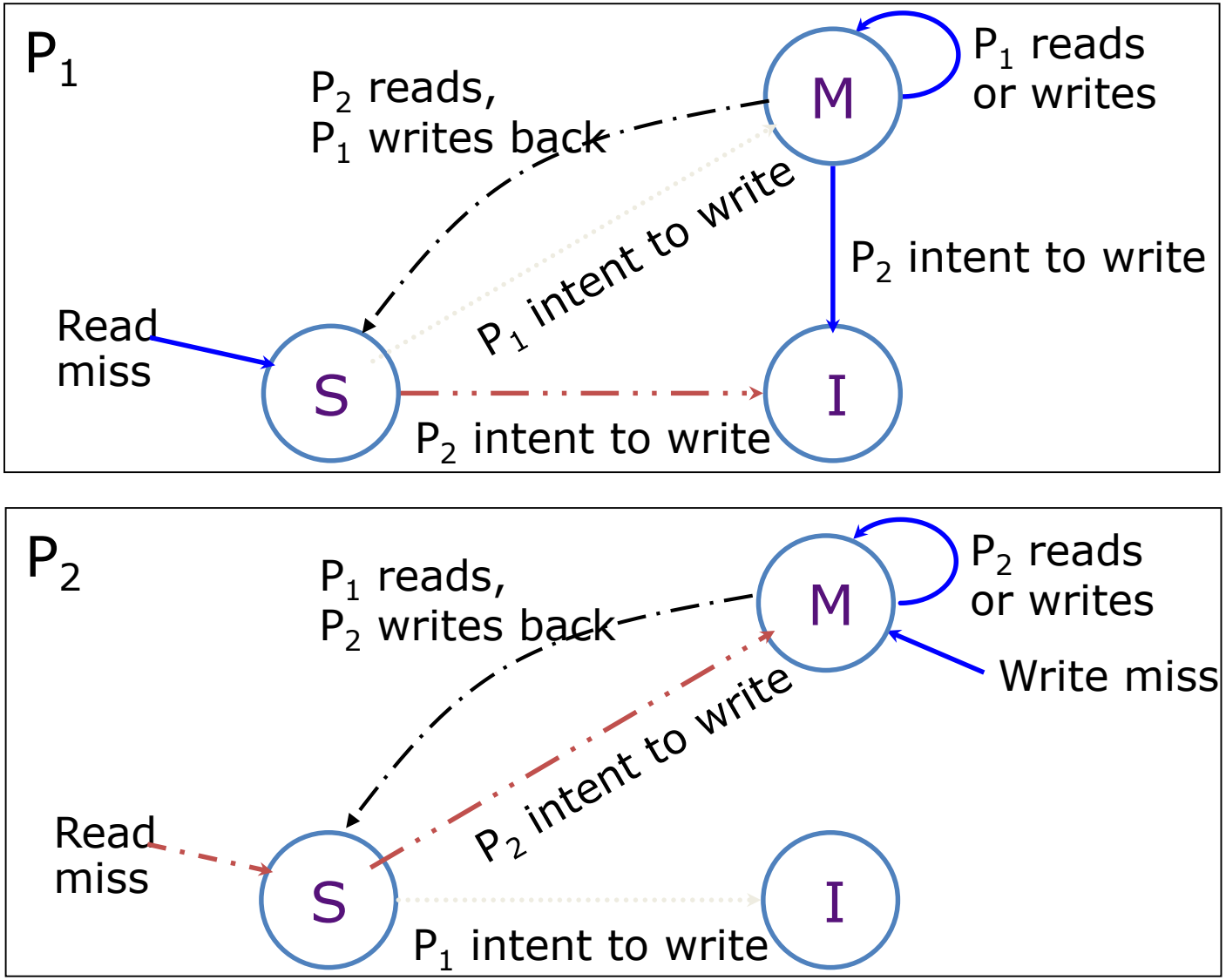
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes



Two Processor Example

(Reading and writing the same cache line)

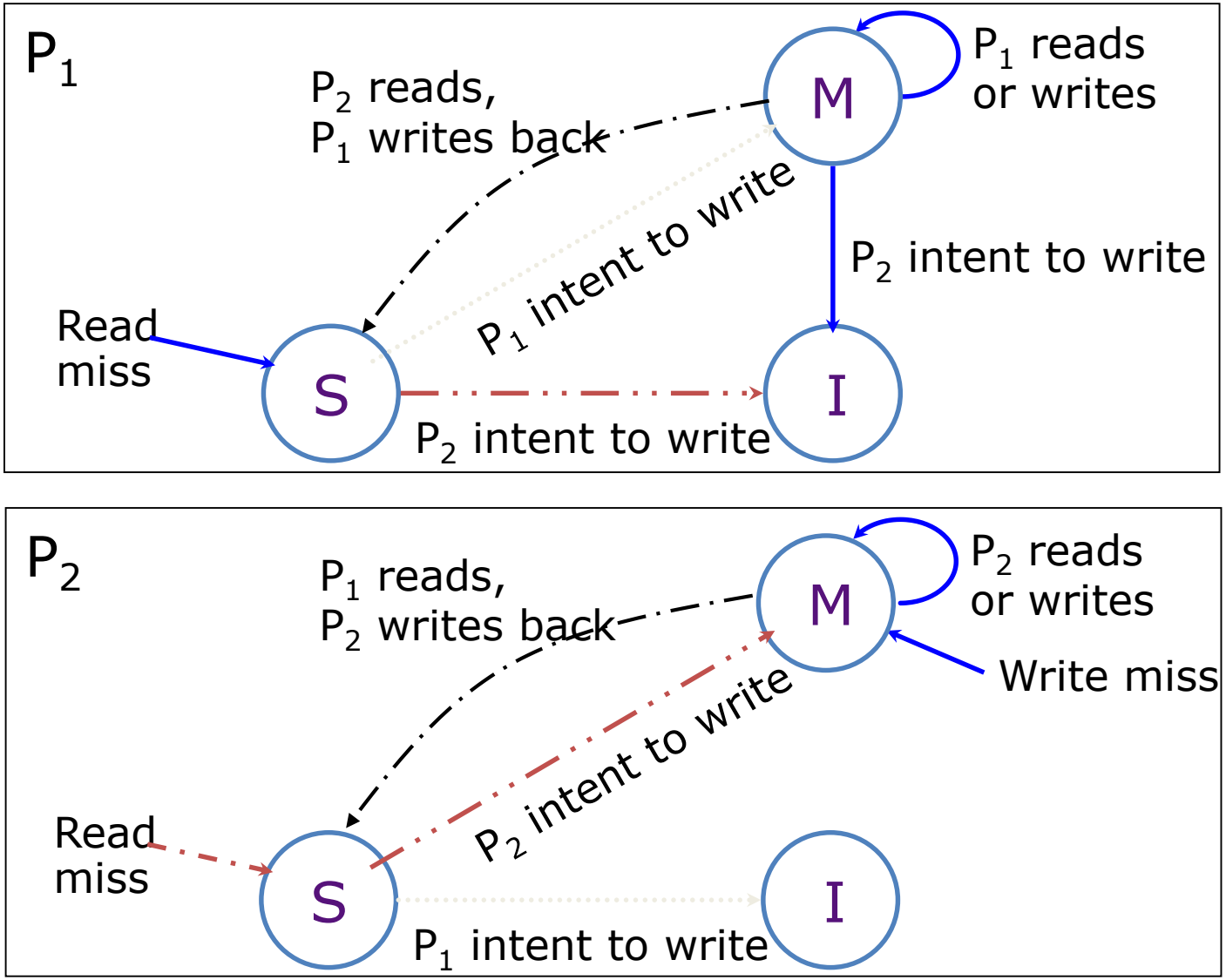
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes



Two Processor Example

(Reading and writing the same cache line)

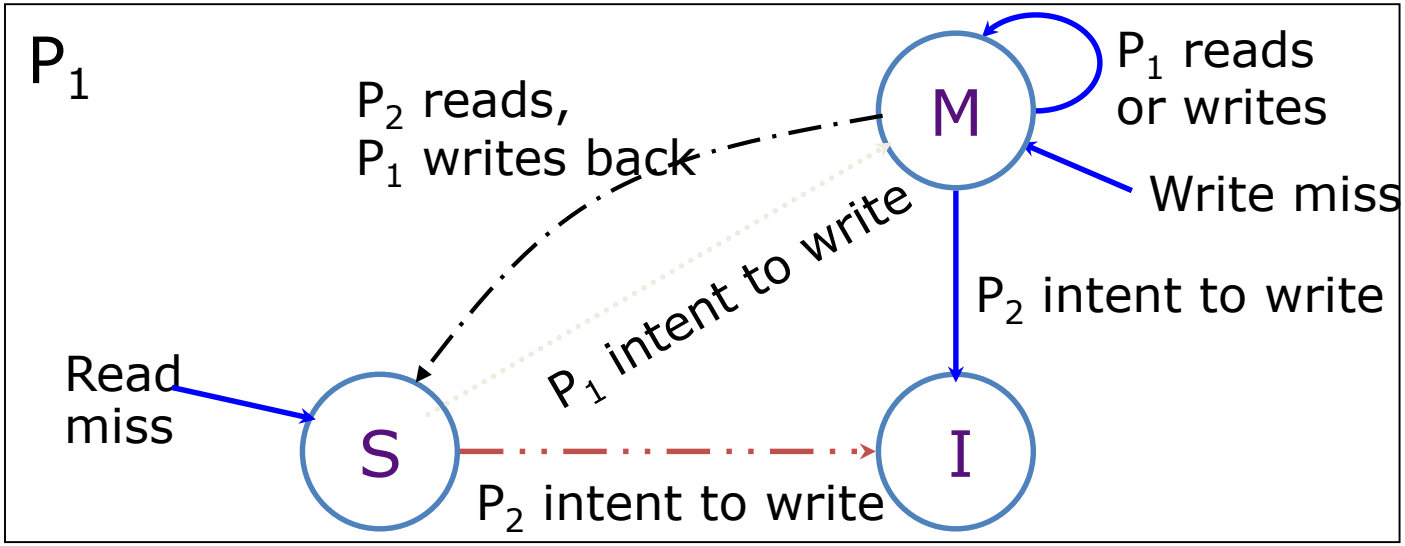
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes

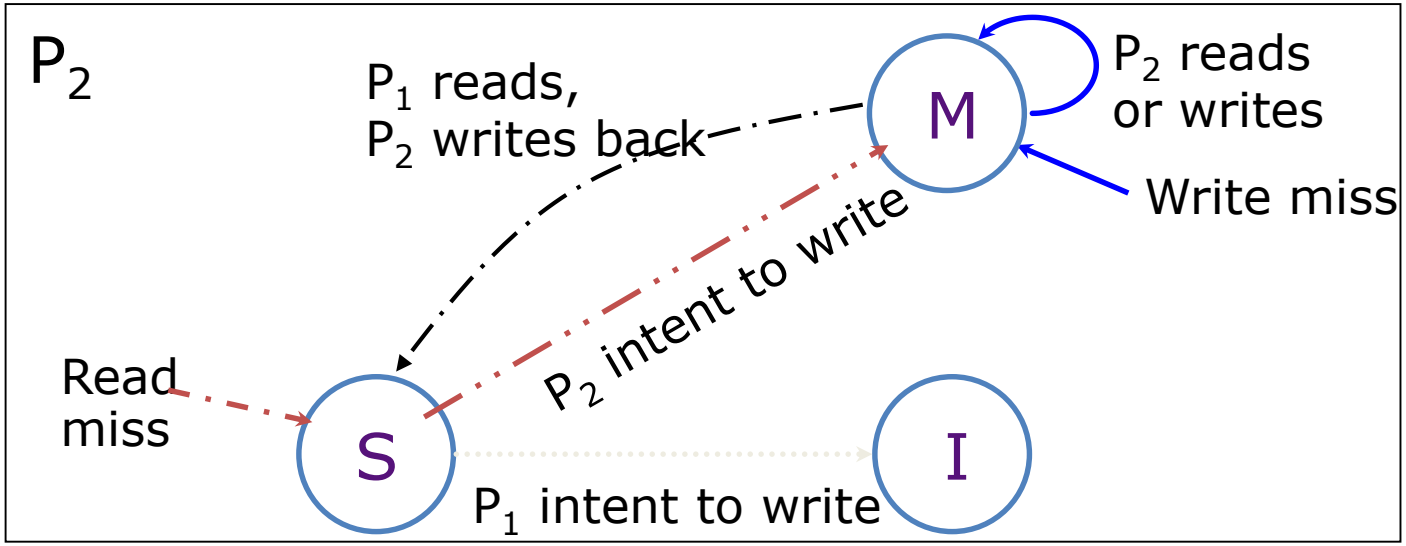


Two Processor Example

(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

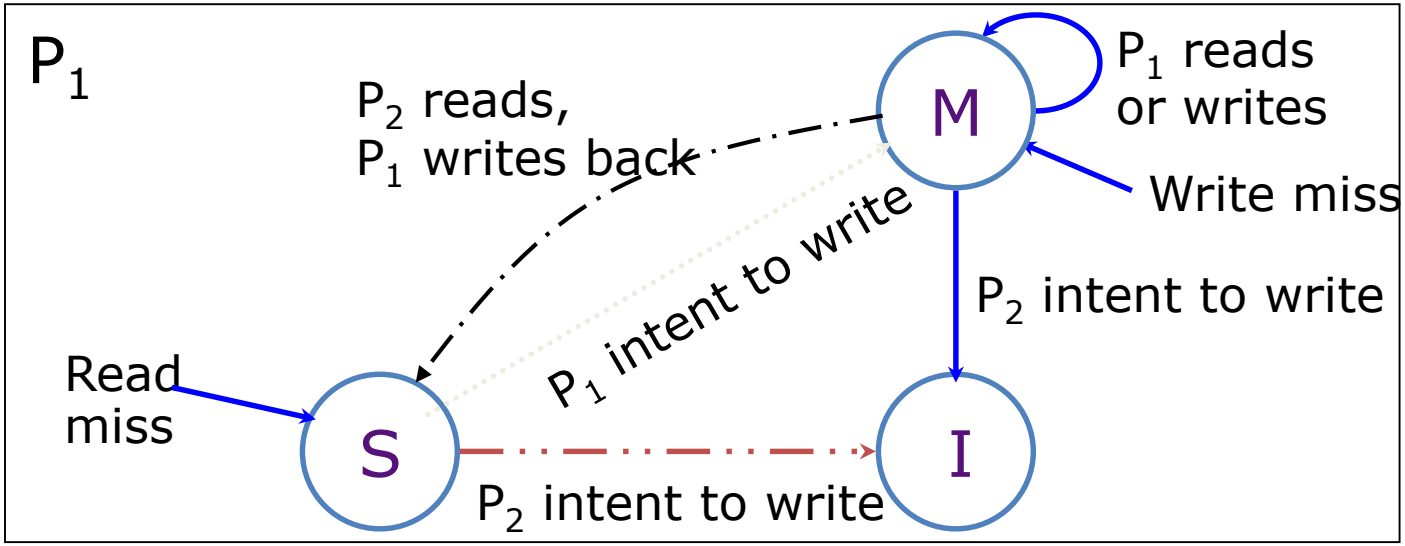


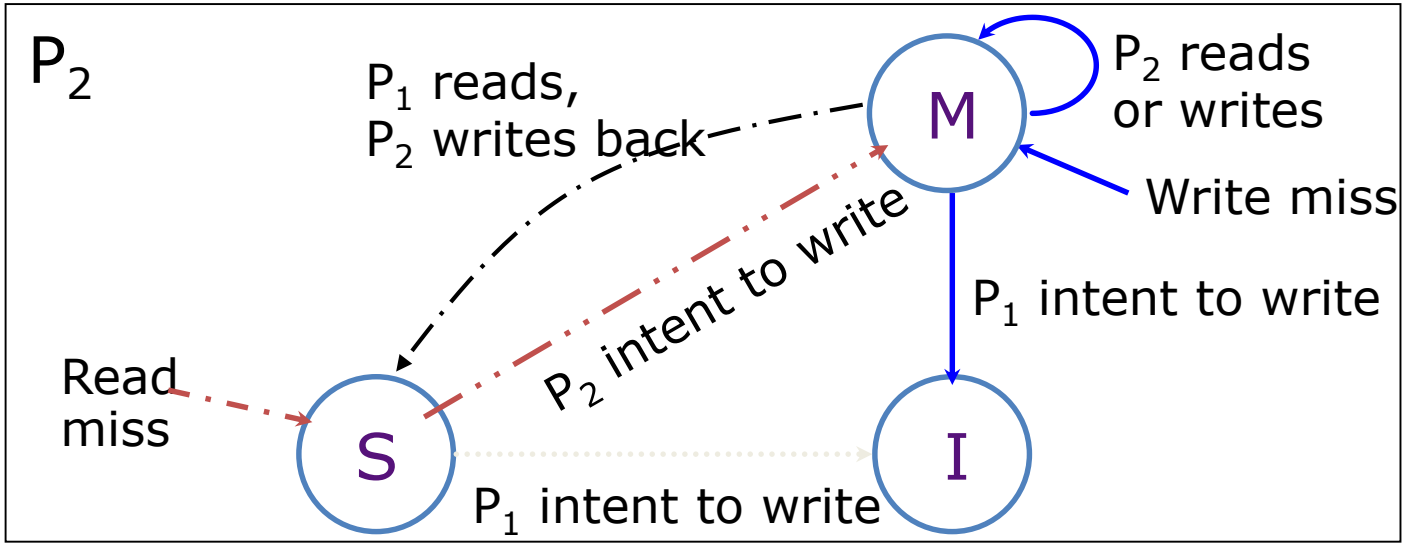


Two Processor Example

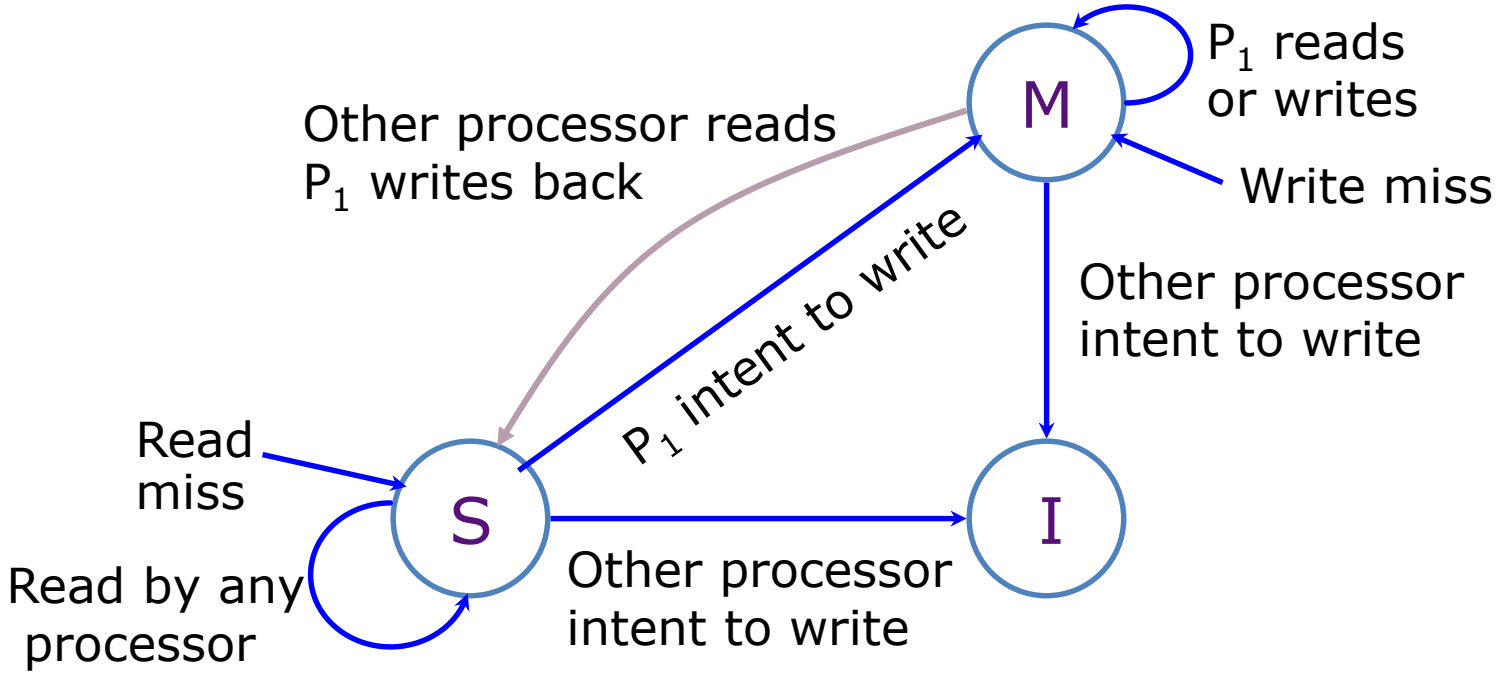
(Reading and writing the same cache line)

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



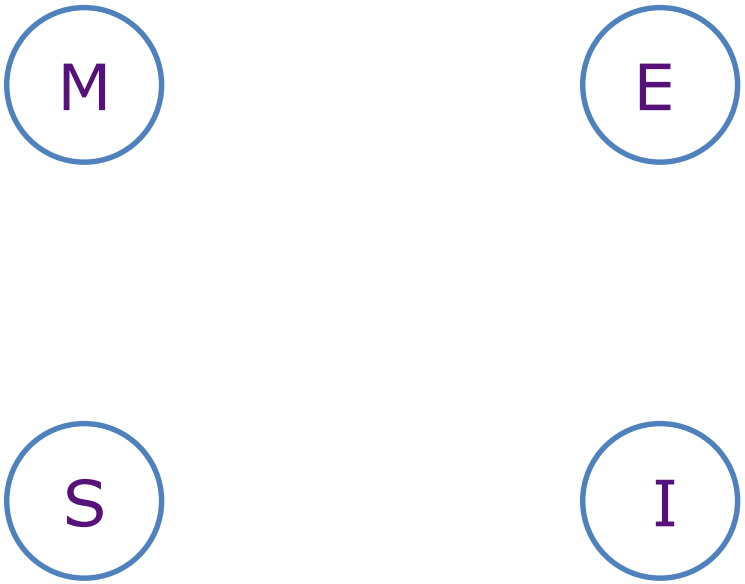
- If a line is in the M state then no other cache can have a copy of the line!
  - Memory stays coherent, multiple differing copies cannot exist

MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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



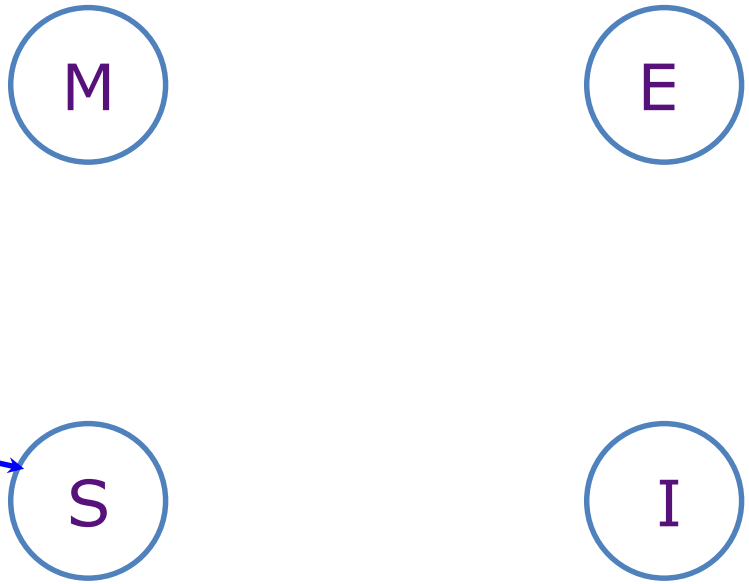
Cache state in processor P_1

MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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



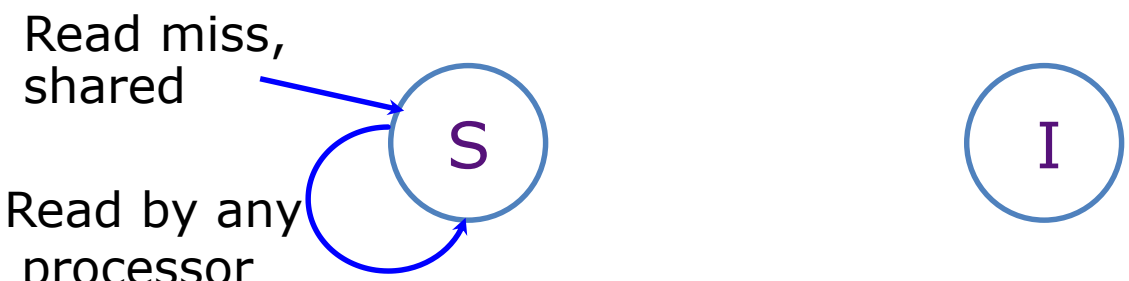
Cache state in processor P_1

MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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



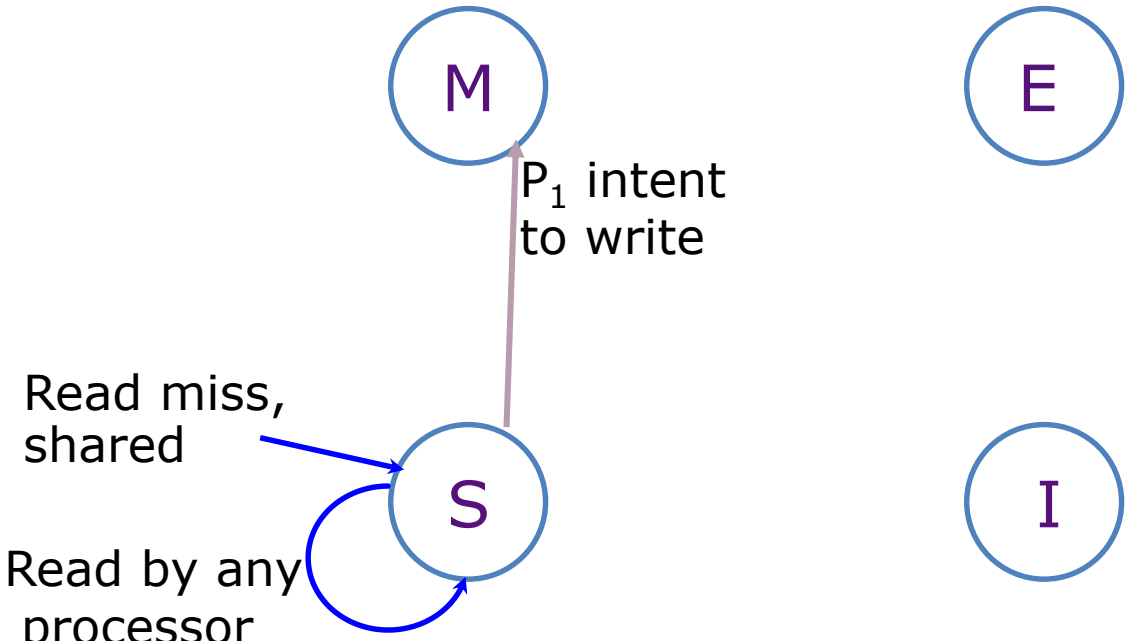
Cache state in processor P_1

MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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



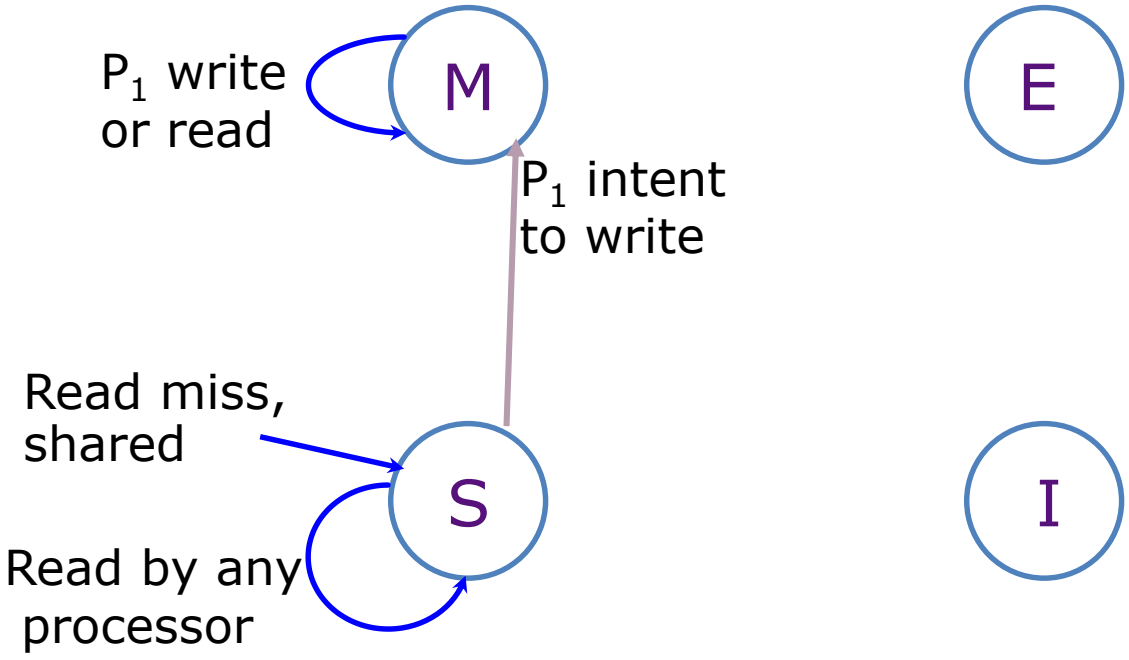
Cache state in processor P_1

MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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



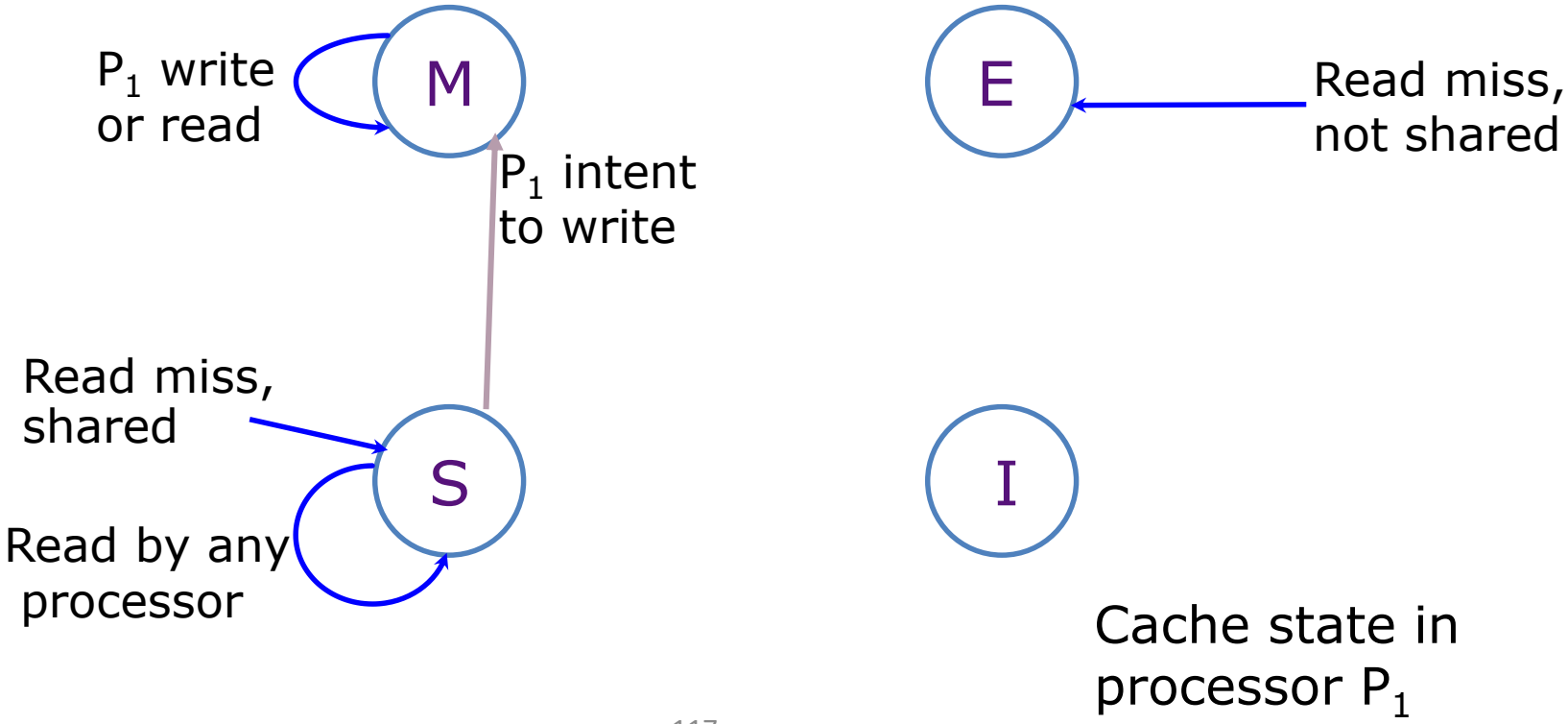
Cache state in processor P_1

MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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

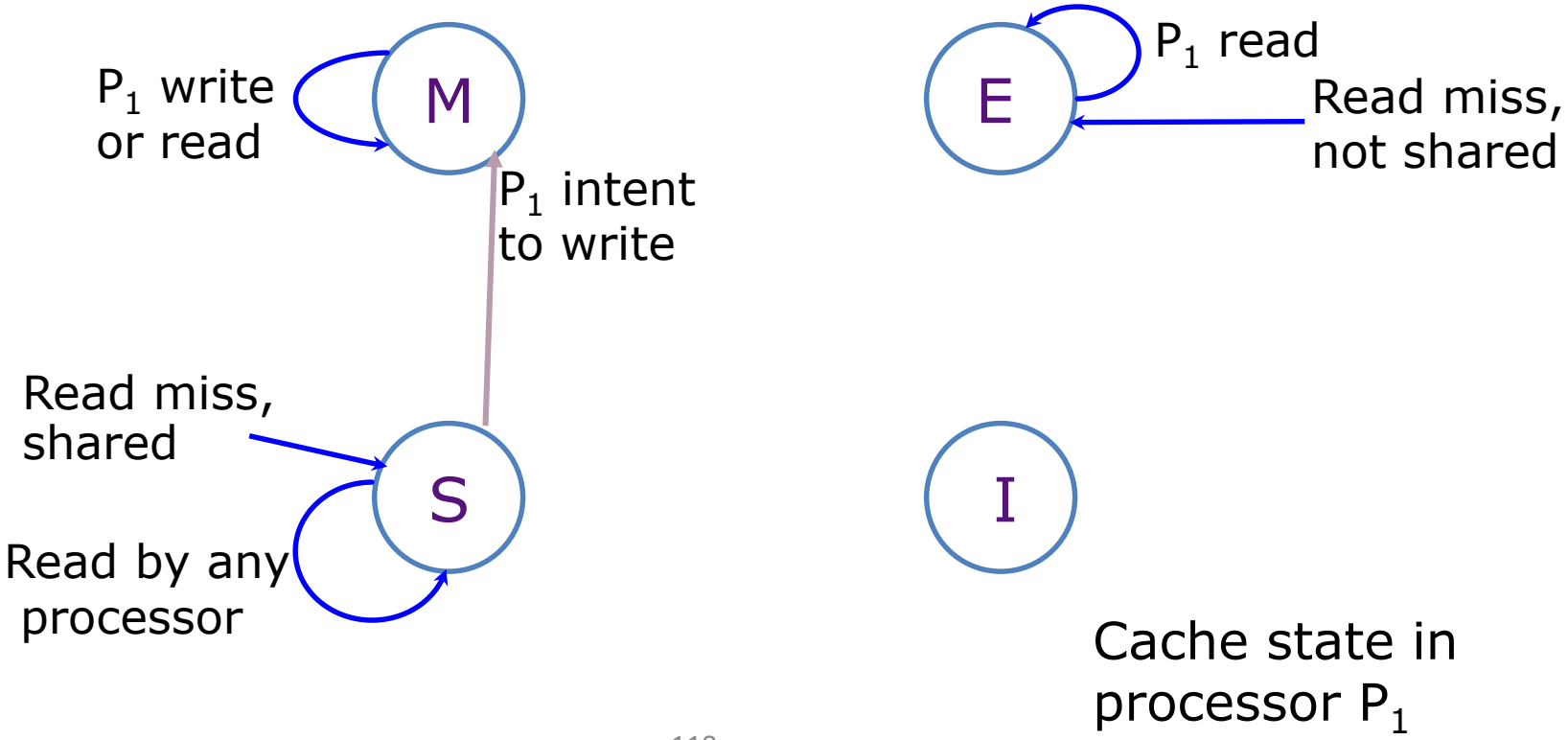


MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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

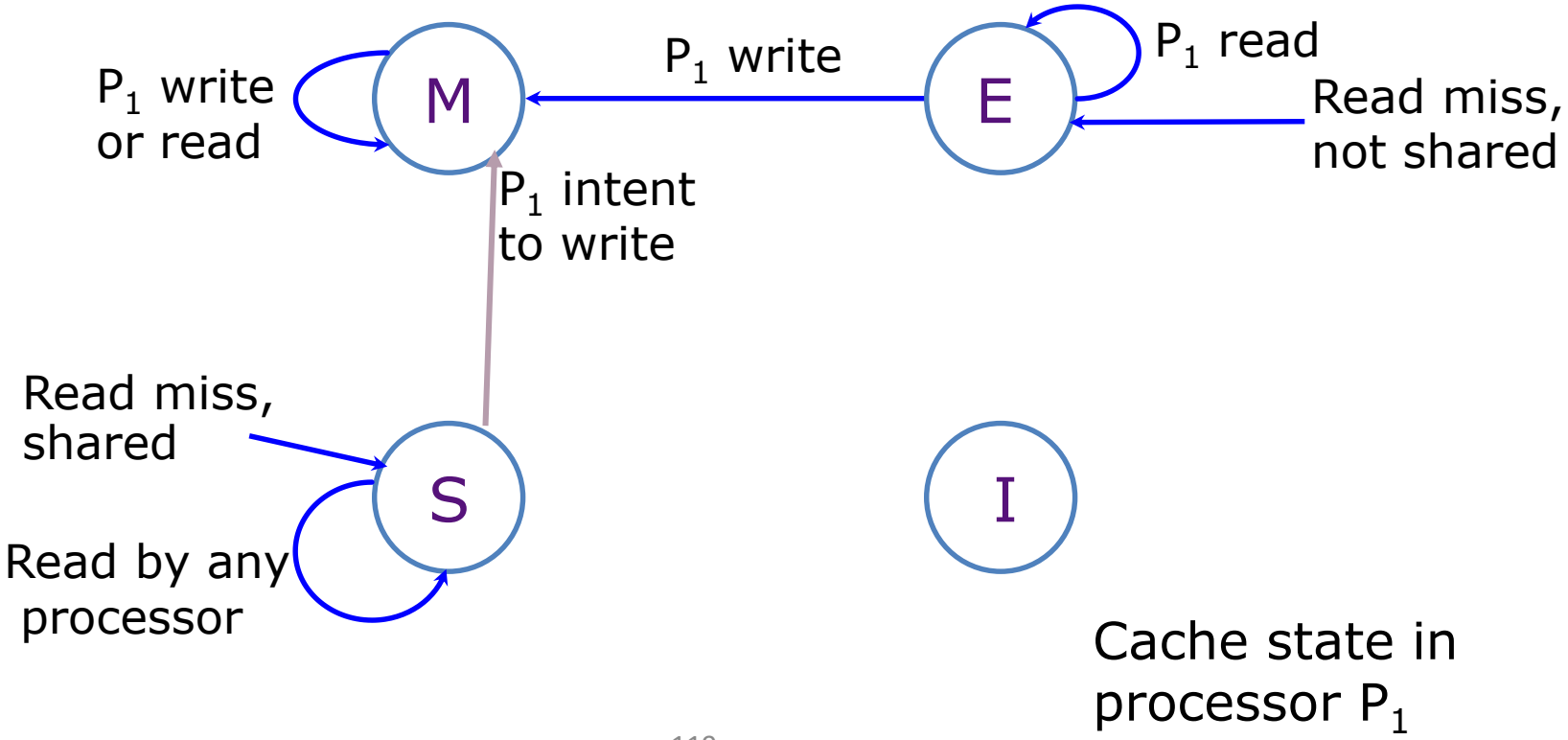


MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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

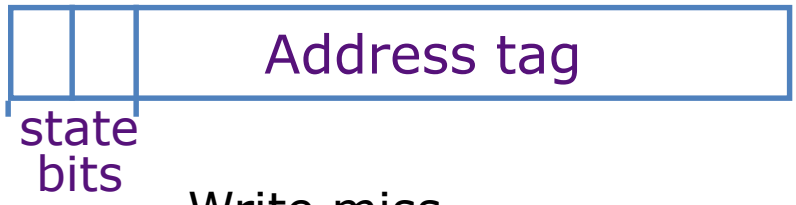


Cache state in processor P_1

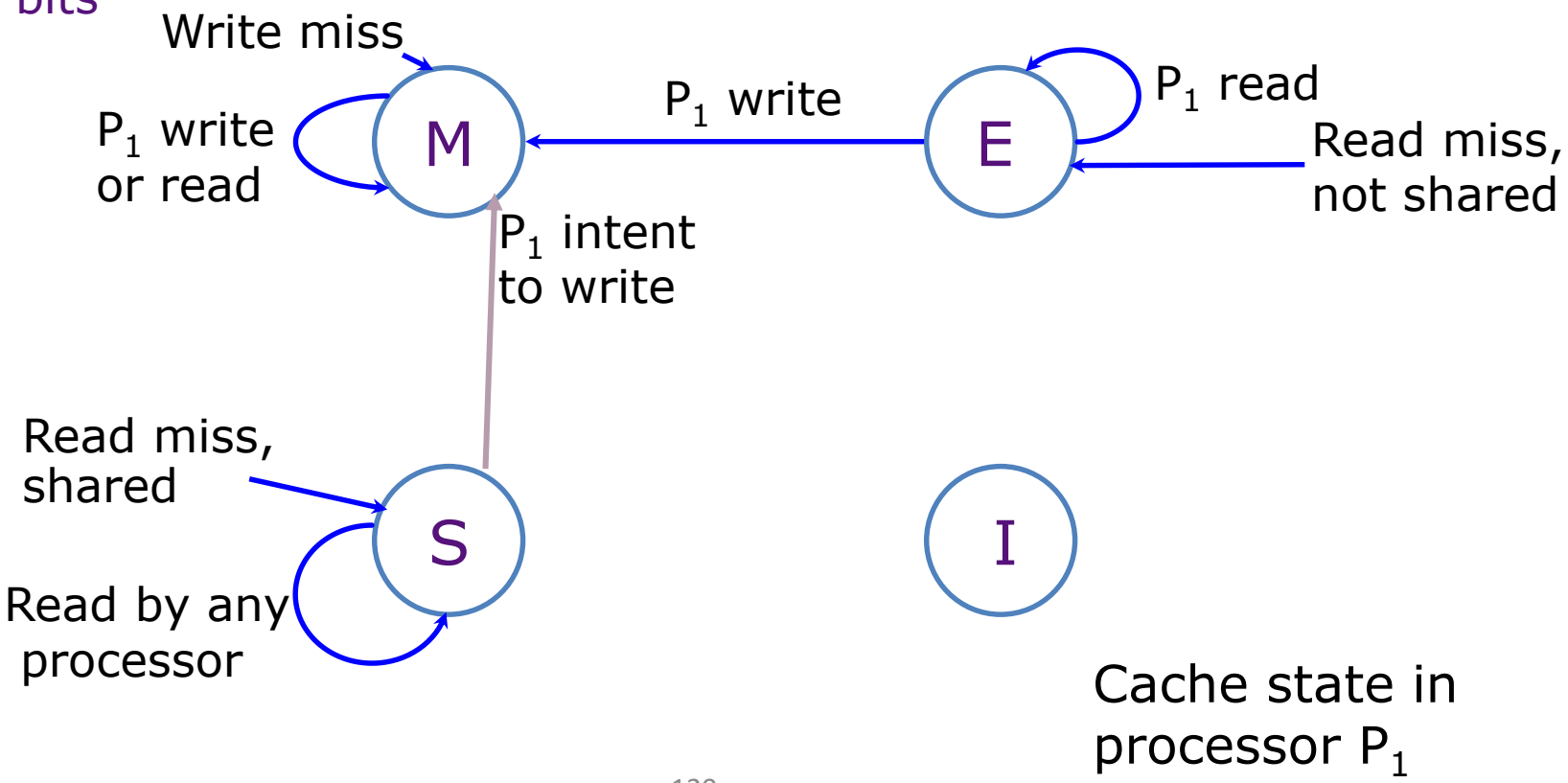
MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag



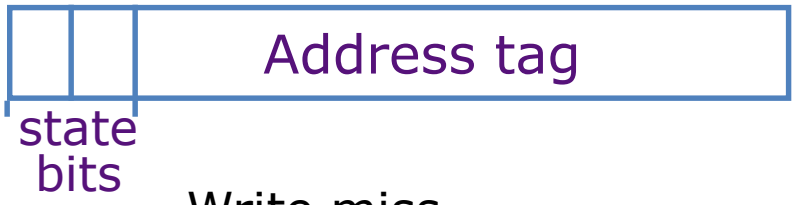
M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



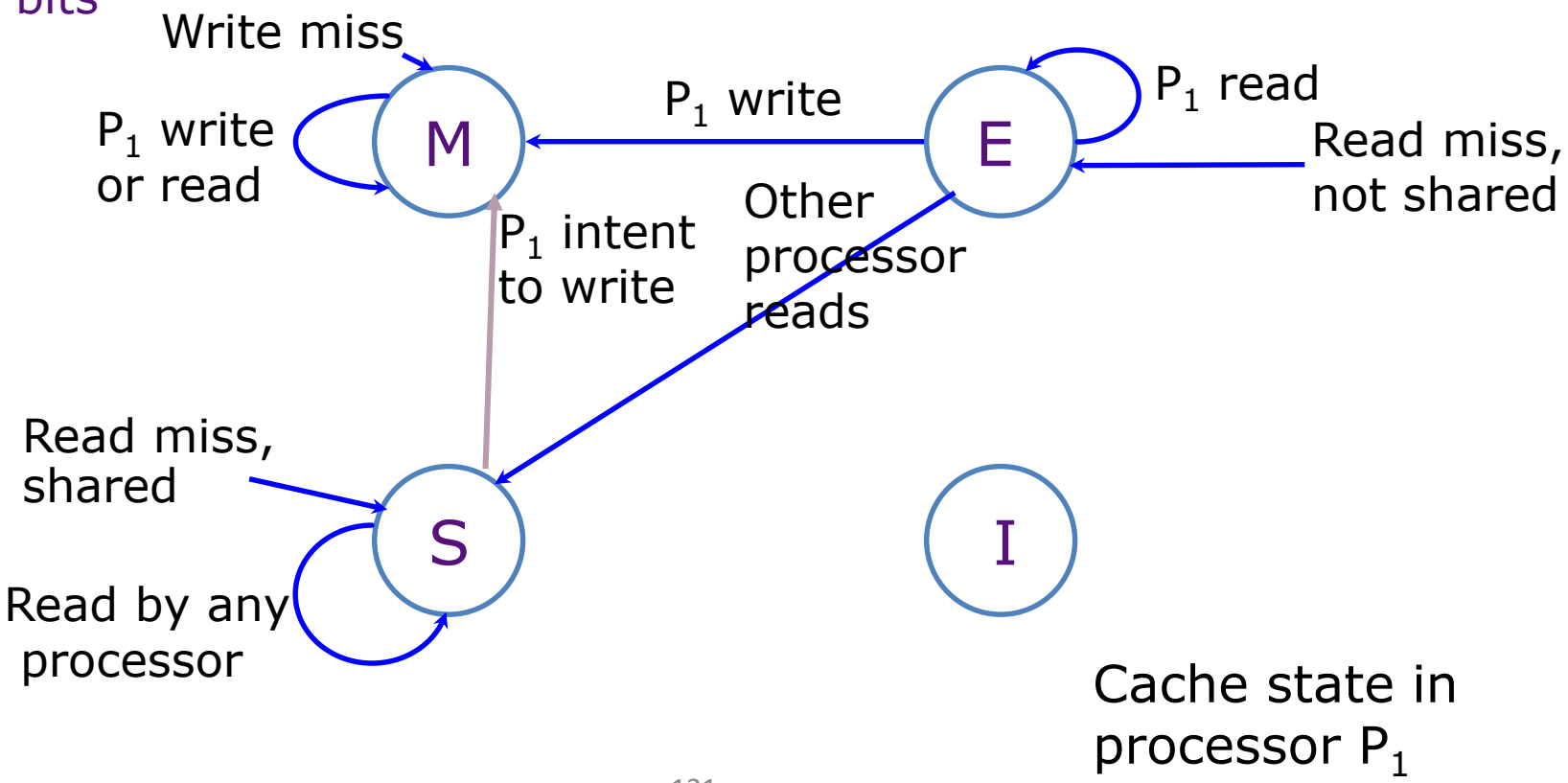
MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag



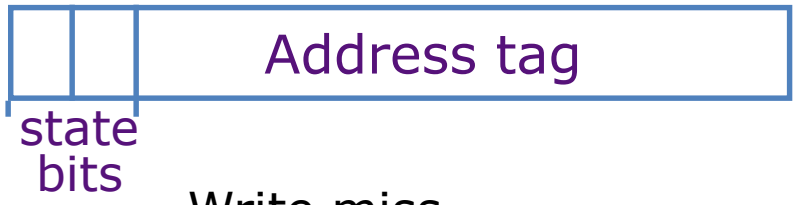
M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



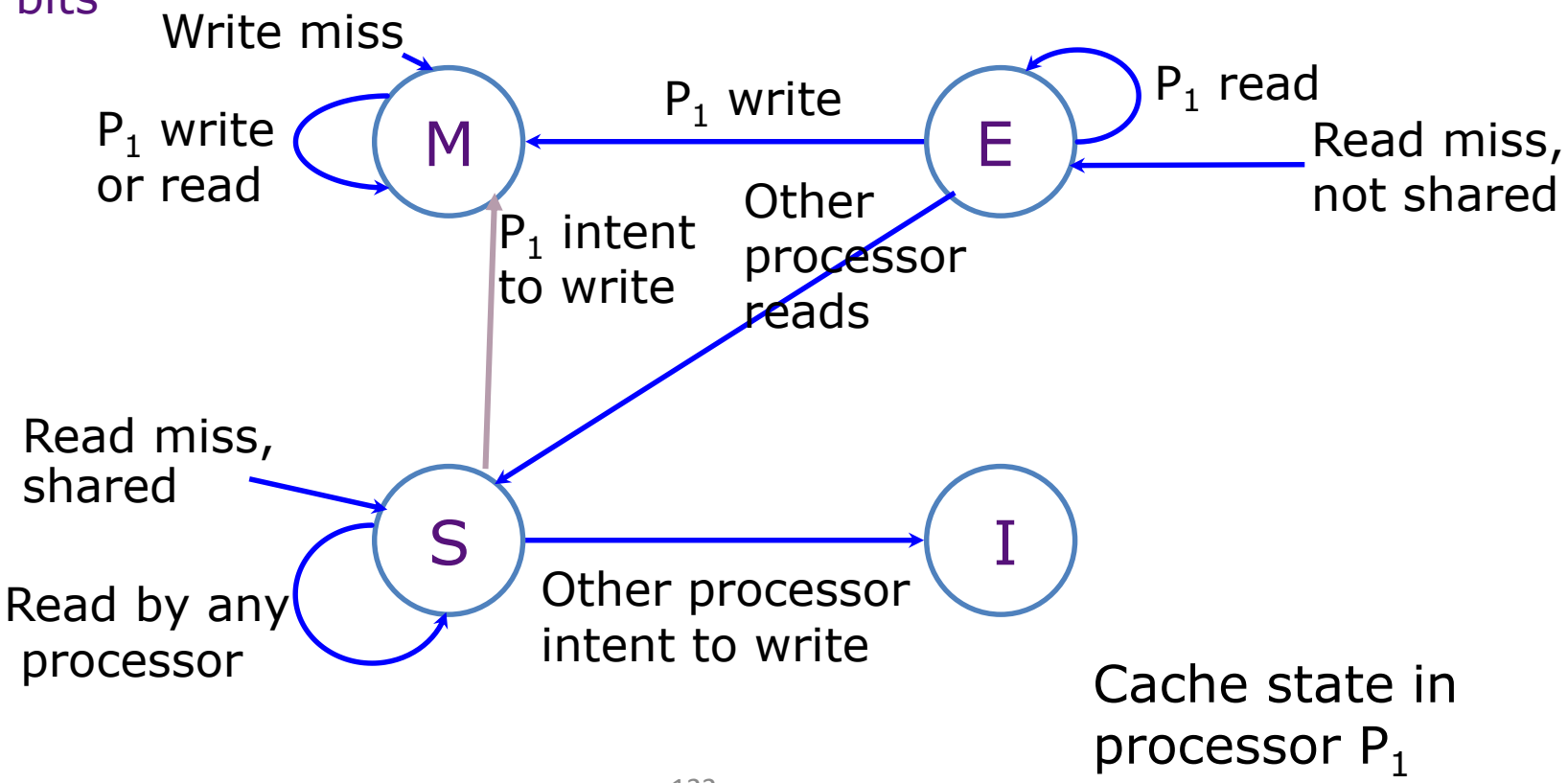
MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag



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

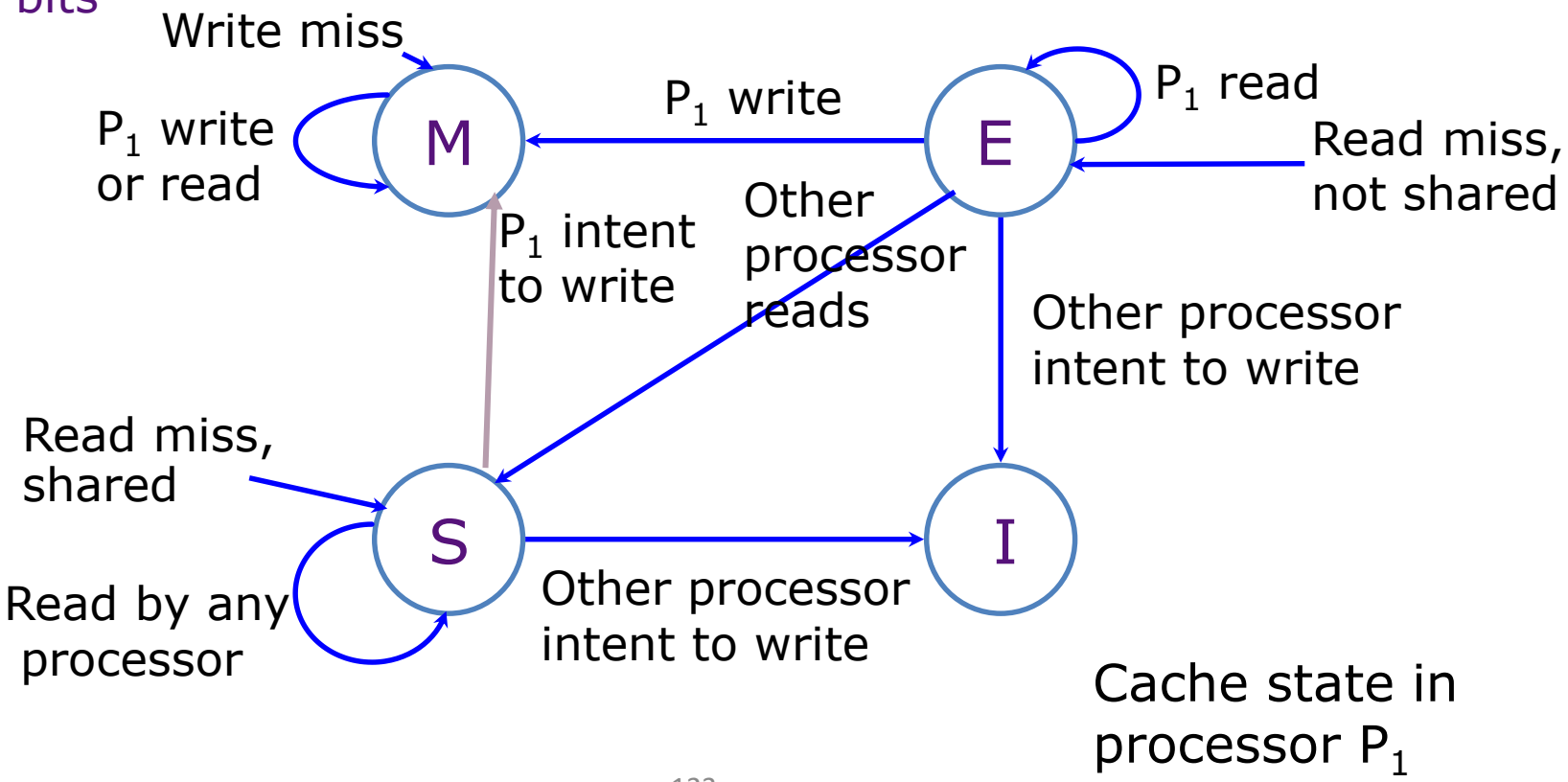


MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

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

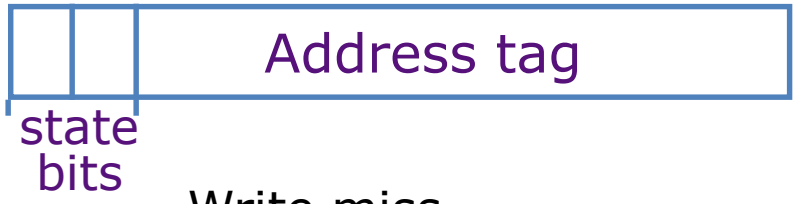


Cache state in processor P_1

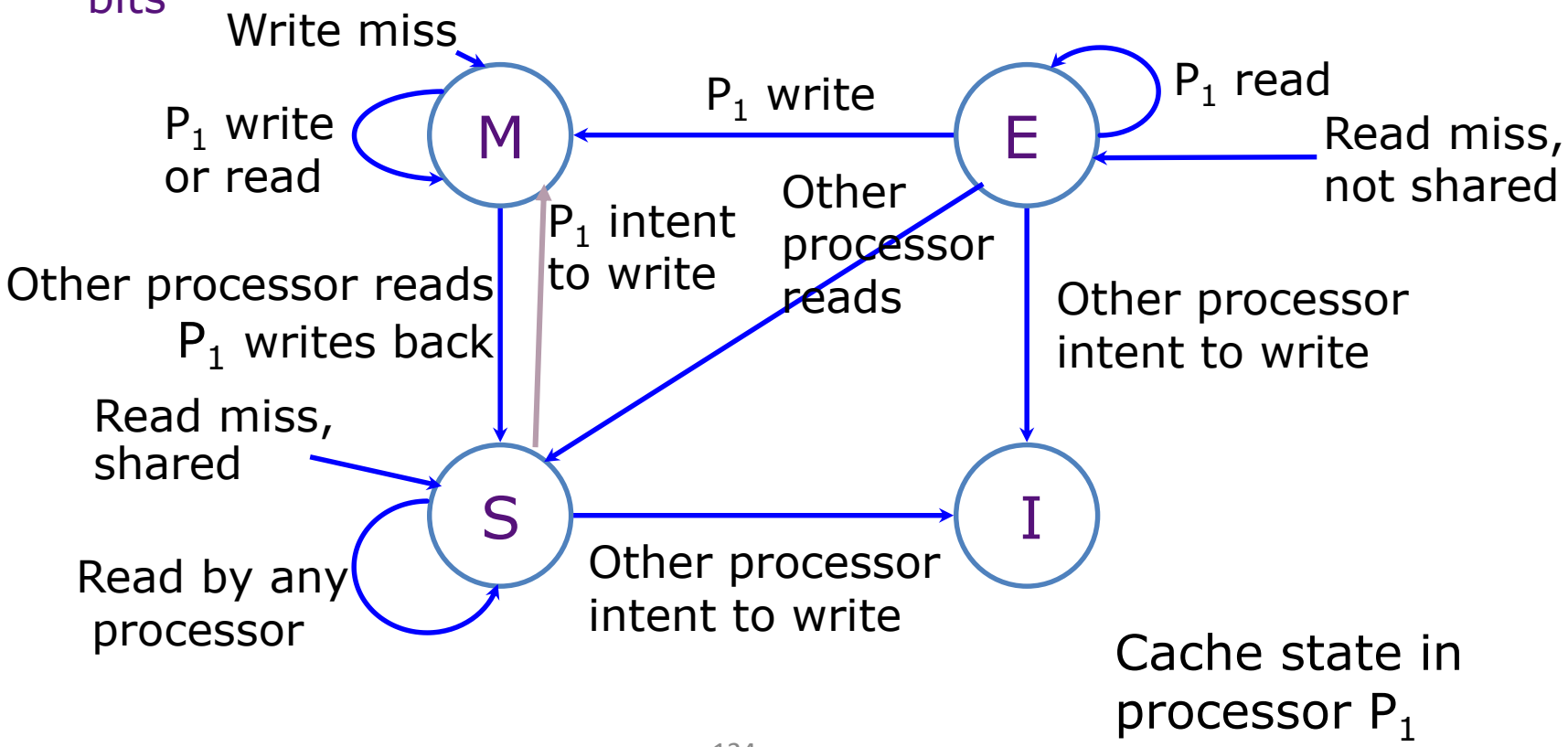
MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag



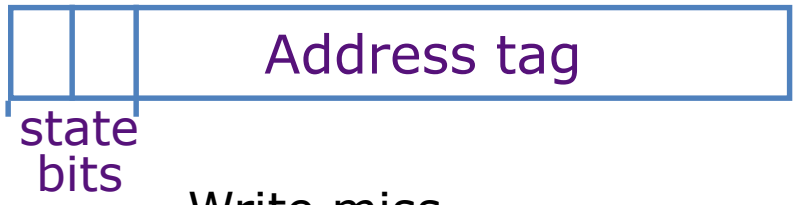
M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



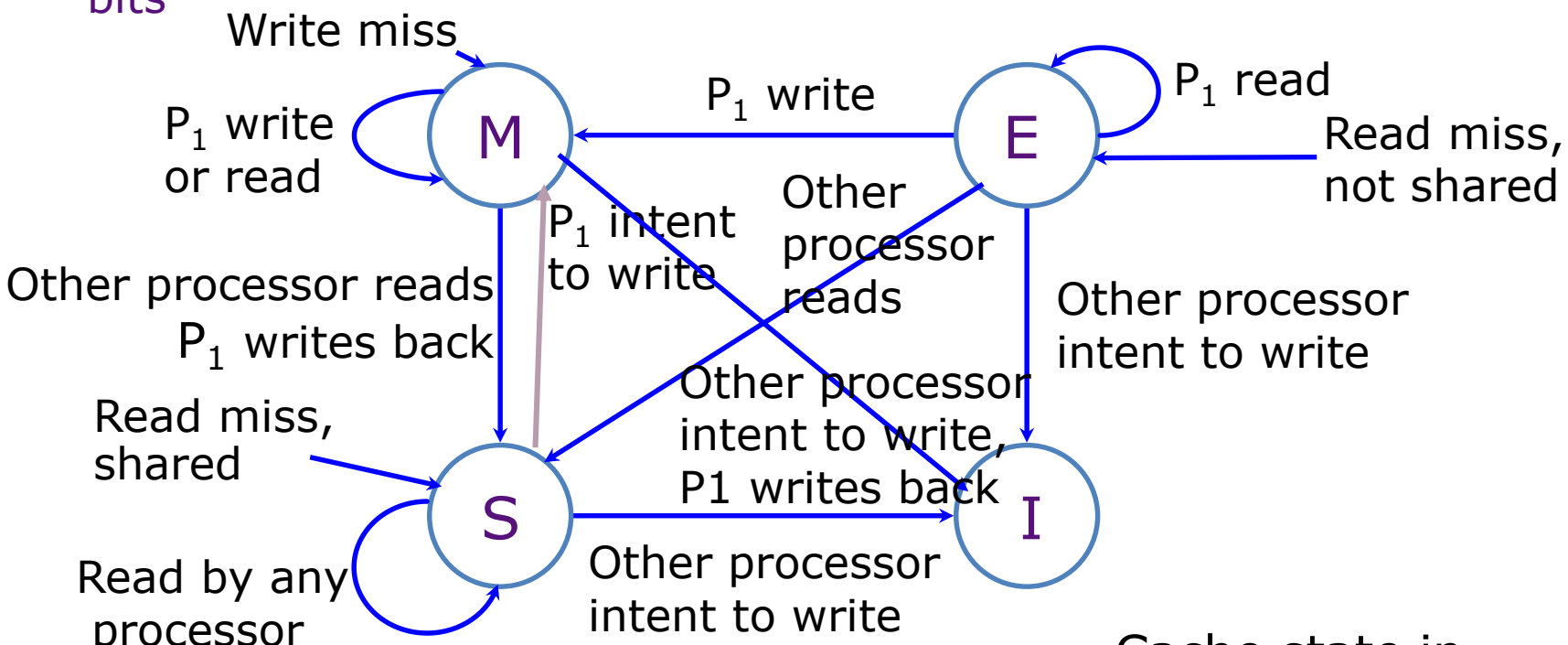
MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag



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



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

- Adds 4th C: *coherence miss*
  - Joins Compulsory, Capacity, Conflict
  - (Sometimes called a *Communication* miss)

Coherency Misses

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
     $\Rightarrow$ miss would not occur if block size were 1 word

False Sharing

A cache block contains more than one word

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

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

What can happen?

Example: True v. False Sharing v. Hit?

- **Assume x1 and x2 in same cache block.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		
2		Read x2	
3	Write x1		
4		Write x2	
5	Read x2		

Example: True v. False Sharing v. Hit?

- **Assume x1 and x2 in same cache block.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	
3	Write x1		
4		Write x2	
5	Read x2		

Example: True v. False Sharing v. Hit?

- **Assume x1 and x2 in same cache block.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		
4		Write x2	
5	Read x2		

Example: True v. False Sharing v. Hit?

- **Assume x1 and x2 in same cache block.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	
5	Read x2		

Example: True v. False Sharing v. Hit?

- **Assume x1 and x2 in same cache block.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		

Example: True v. False Sharing v. Hit?

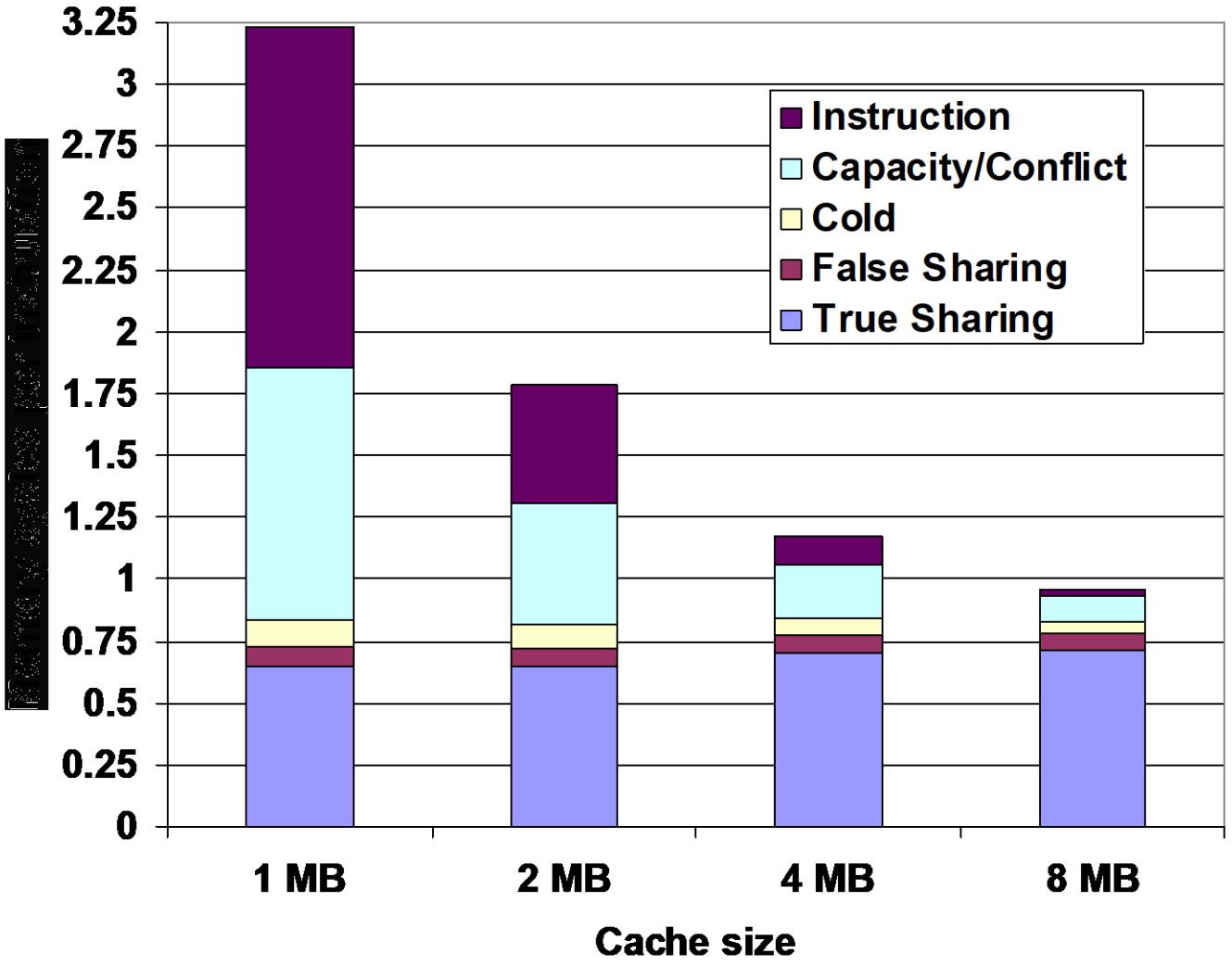
- **Assume x1 and x2 in same cache block.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

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

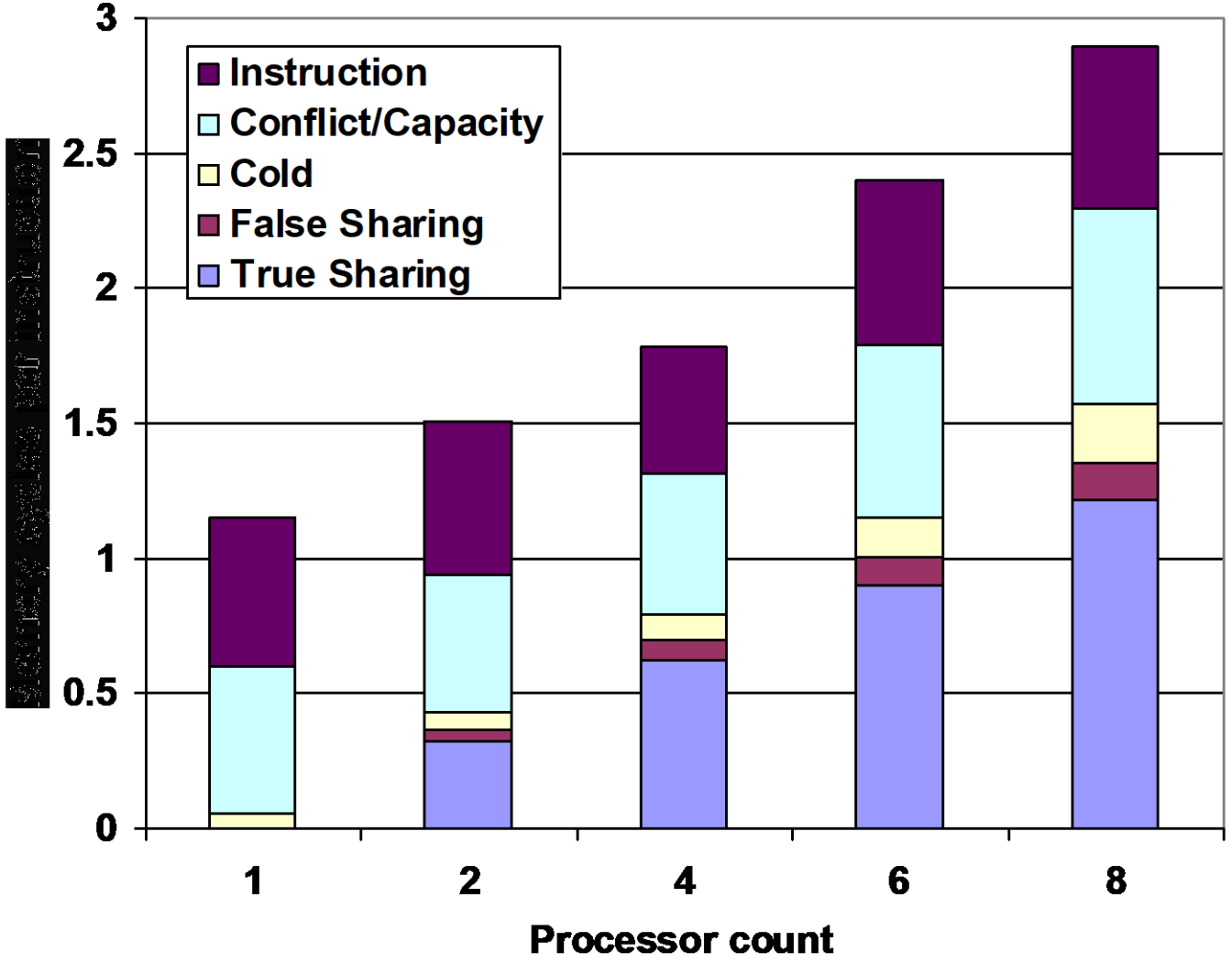
- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)

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



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

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



A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (0) Determine when to invoke coherence protocol
  - (a) Find info about state of address in other caches to determine action
    - whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (invalidate/update)
- (0) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an "access fault" occurs on the line
- Different approaches distinguished by (a) to (c)

Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a "search"
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn't scale with number of processors, P
  - on bus, bus bandwidth doesn't scale
  - on scalable network, every fault leads to at least P network transactions
- Scalable coherence:
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol

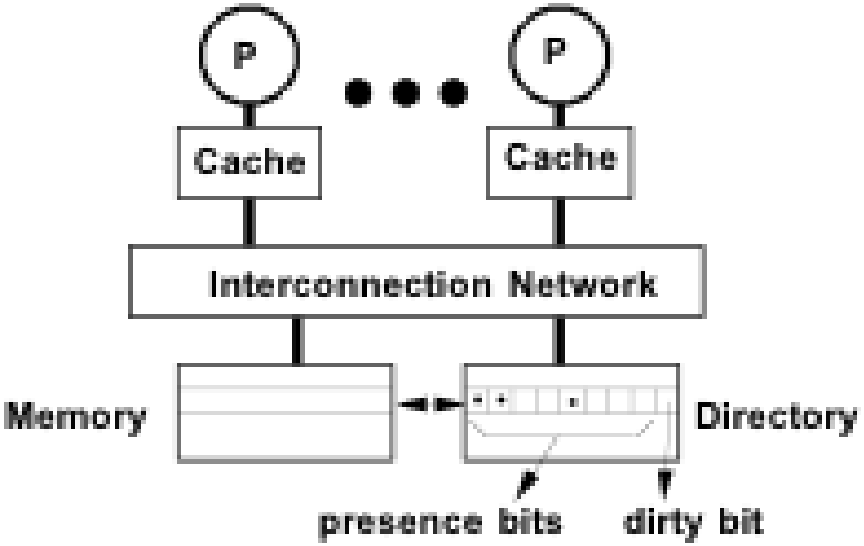
Need for a more scalable protocol

- Snoopy schemes do not scale because they rely on broadcast
- Hierarchical snoopy schemes have the root as a bottleneck
- **Directory** based schemes allow scaling
  - They avoid broadcasts by keeping track of all CPUs caching a memory block, and then using point-to-point messages to maintain coherence
  - They allow the flexibility to use any scalable point-to-point network

Scalable Approach: Directories

- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Basic Operation of Directory



- k processors.
- With each cache-block in memory:
  k presence-bits, 1 dirty-bit
- With each cache-block in cache:
  1 valid bit, and 1 dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit ON then { recall line from dirty proc (downgrade cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}
- Write to main memory by processor i:
  - If dirty-bit OFF then {send invalidations to all caches that have the block; turn dirty-bit ON; supply data to i; turn p[i] ON; ... }

Approfondimento

- Si vedano le slides 13-29 della lezione 12 «Directory-based cache-coherence» dal corso "CMU 15-418 - Parallel Computer Architecture and Programming" di Kayvon Fatahalian, Carnegie Mellon University
  - Caricate sul sito

Three fundamental issues for shared memory multiprocessors

- **Coherence**
  Do I see the most recent data?

- **Synchronization**
  How to synchronize processes?
  – how to protect access to shared data?

- **Consistency**
  ***When** do I see a written value?*
  – e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

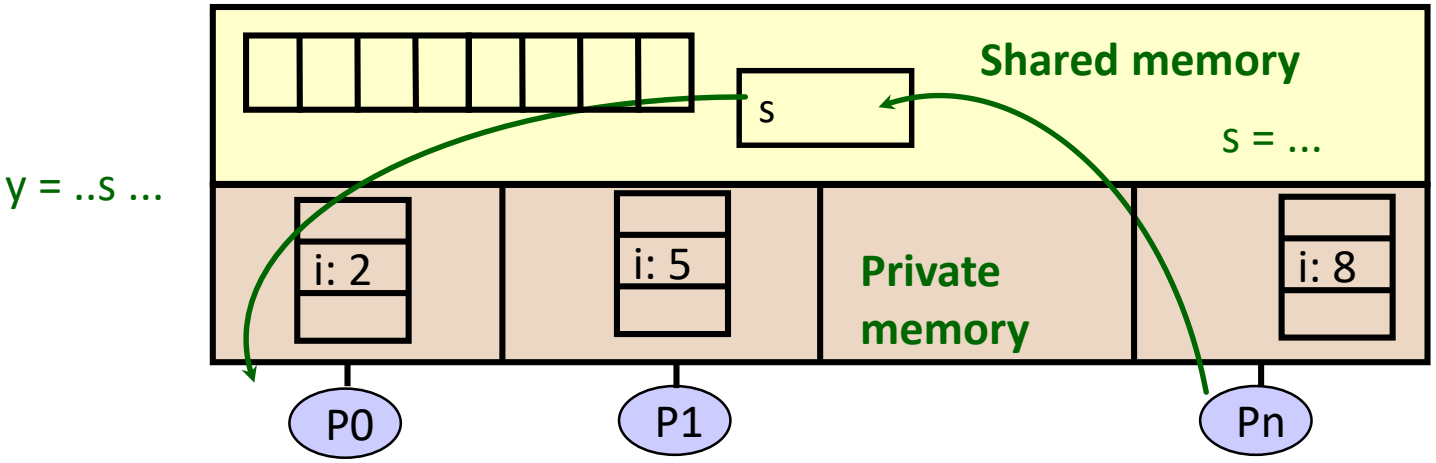
Motivation: “Too much milk”

- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Shared Memory Synchronization

- 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



Shared Memory code for computing a sum

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

```
Thread 1

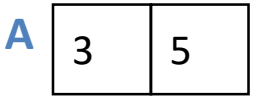
  for i = 0, n/2-1
      s = s + sqr(A[i])
```

```
Thread 2

  for i = n/2, n-1
      s = s + sqr(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

Shared Memory code for computing a sum



f = square

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

Thread 1

```
....
 compute f([A[i]) and put in reg0
 reg1 = s
 reg1 = reg1 + reg0
 s = reg1
...
```

Thread 2

```
...
 compute f([A[i]) and put in reg0
  reg1 = s
  reg1 = reg1 + reg0
  s = reg1
...
```

Shared Memory code for computing a sum

A | 3 | 5 |

f = square

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

Thread 1

```
....
compute f([A[i]) and put in reg0
reg1 = s
reg1 = reg1 + reg0
s = reg1
…
```

9

Thread 2

```
…
compute f([A[i]) and put in reg0
reg1 = s
reg1 = reg1 + reg0
s = reg1
…
```

Shared Memory code for computing a sum

A | 3 | 5 |

f = square

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

Thread 1

```
....
compute f([A[i]) and put in reg0
reg1 = s
reg1 = reg1 + reg0
s = reg1
...
```

9

Thread 2

```
...
compute f([A[i]) and put in reg0
reg1 = s
reg1 = reg1 + reg0
s = reg1
...
```

25

Shared Memory code for computing a sum

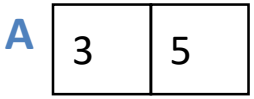
A | 3 | 5 |

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	
reg1 = reg1 + reg0		reg1 = reg1 + reg0	
s = reg1		s = reg1	
…		…	

Shared Memory code for computing a sum



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		reg1 = reg1 + reg0	
s = reg1		s = reg1	
…		…	

Shared Memory code for computing a sum

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		reg1 = reg1 + reg0	25
s = reg1		s = reg1	
…		…	

Shared Memory code for computing a sum

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		reg1 = reg1 + reg0	25
s = reg1		s = reg1	25
…		…	

Shared Memory code for computing a sum

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		s = reg1	25
…		…	

Shared Memory code for computing a sum

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
…		…	

Shared Memory code for computing a sum

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 no += operation is not atomic
  - All computations happen in (private) registers

Shared Memory code for computing a sum

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

Thread 1

```
local_s1= 0
for i = 0, n/2-1
   local_s1 = local_s1 + sqr(A[i])

s = s + local_s1
```

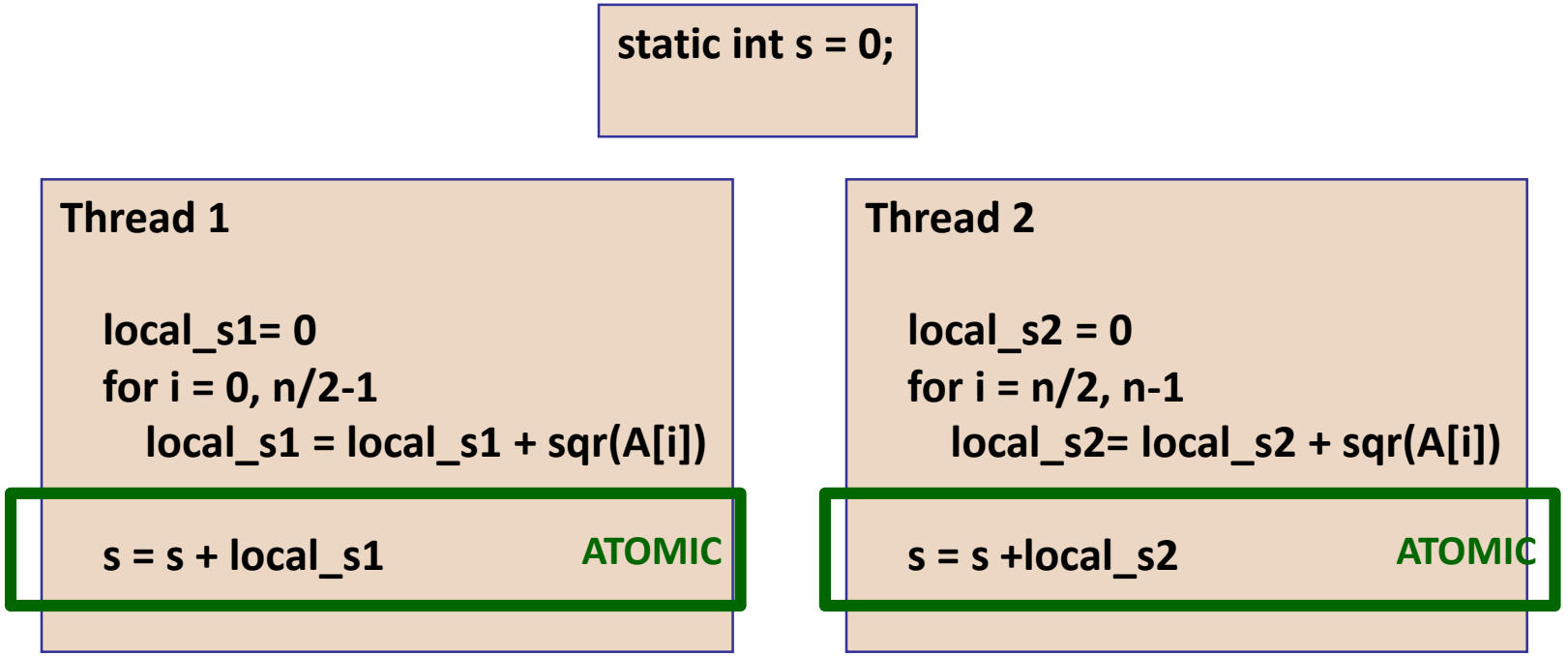
Thread 2

```
local_s2 = 0
for i = n/2, n-1
   local_s2= local_s2 + sqr(A[i])

s = s +local_s2
```

- Since addition is associative, it's OK to rearrange order
  - Right?
- 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

Shared Memory code for computing a sum



- Since addition is associative, it's OK to rearrange order
  - Right?
- 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

Atomic Operations

- To understand a concurrent program, we need to know what the underlying **indivisible operations** are!

- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together

- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - hard to build anything useful with only reads and writes

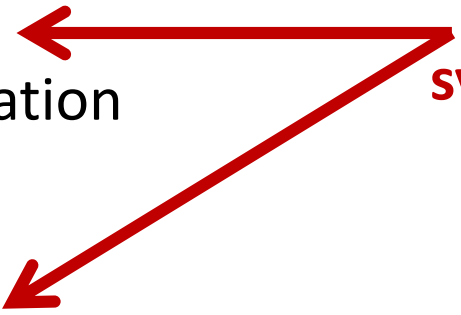
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task

- Critical Section: piece of code that only one thread can execute at once
  - Critical section and mutual exclusion are two ways of describing the same thing
  - Critical section defines sharing granularity

Role of Synchronization

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

- Types of Synchronization
  - *Mutual Exclusion*
  - Event synchronization
    - *point-to-point*
    - group
    - *global (barriers)*

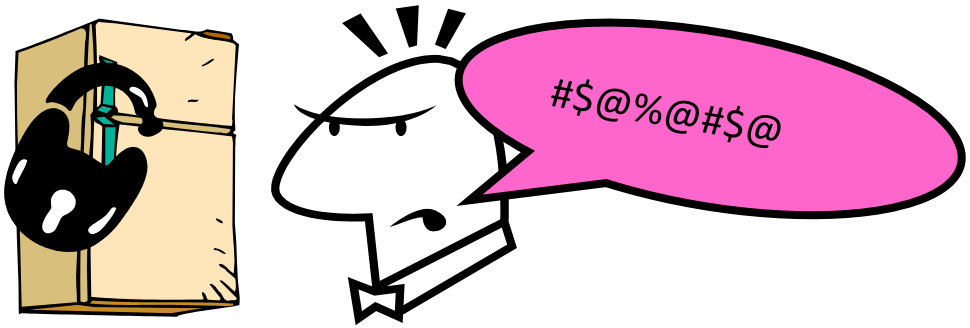


Most used forms of synchronization in shared memory parallel programming

- How much hardware support?

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
- Example: fix the milk problem by putting a lock on refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much (coarse granularity): roommate angry if only wants orange juice



Too Much Milk: Correctness properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down **desired** behavior first
  - think first, then code

- What are the correctness properties for the “Too much milk” problem?
  - Never more than one person buys
  - Someone buys if needed

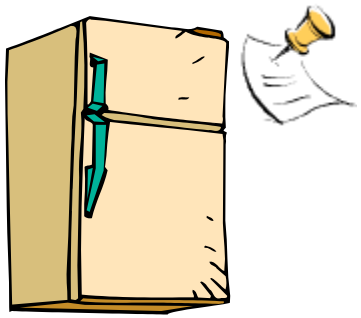
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)

- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove note;
    }
}
```



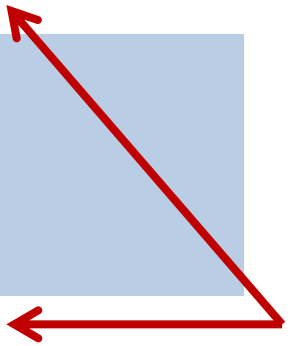
- Result?

Too Much Milk: Solution #1

```
  Thread A                        Thread B
if (noMilk)
  if (noNote) {
                              if (noMilk)
                                if (noNote) {
    leave Note;
     buy milk;
    remove note;
  }
}
                                   leave Note;
                                    buy milk;
                                   remove note;
                                 }
                               }
```

Too Much Milk: Solution #1

```
  Thread A                          Thread B
if (noMilk)
  if (noNote) {
                                  if (noMilk)
                                    if (noNote) {
    leave Note;
     buy milk;
    remove note;
  }
}
                                     leave Note;
                                      buy milk;
                                     remove note;
                                   }
                                  }
```



Need to atomically update lock variable

How to Implement Lock?

- Lock: prevents someone from accessing something
  - Lock before entering critical section (e.g., before accessing shared data)
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
    - Should sleep if waiting for long time

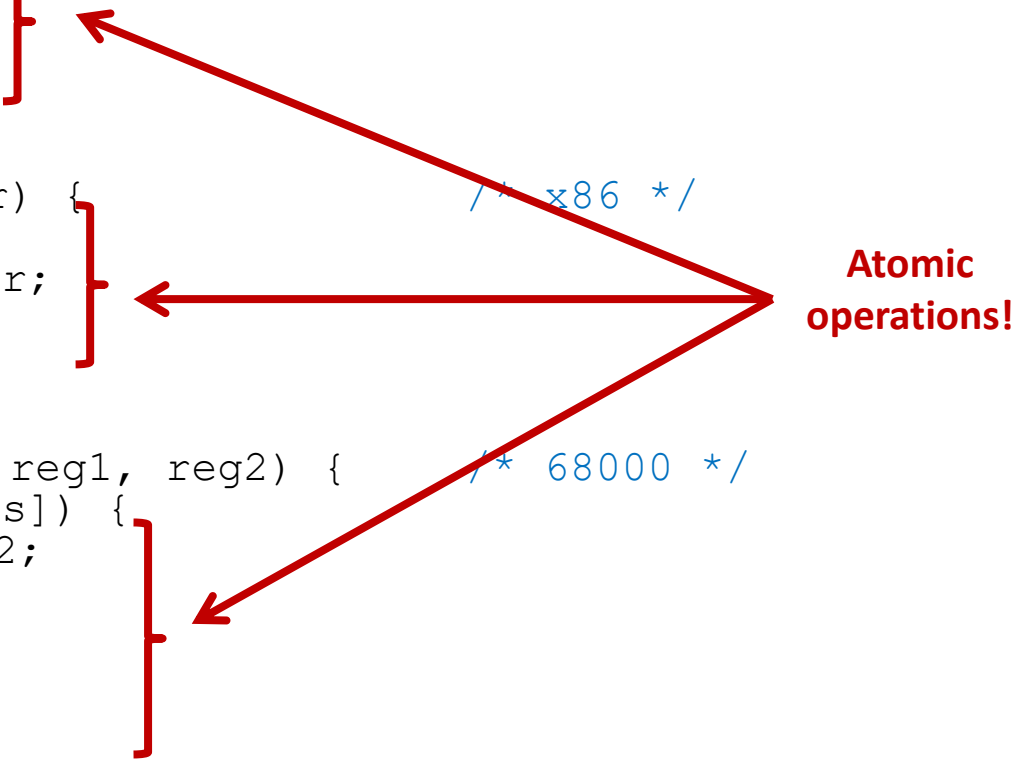
- Hardware atomic instructions?

Examples of hardware atomic instructions

- ```
  test&set (&address) {
      result = M[address];
      M[address] = 1;
      return result;
  }
  ```
  /* most architectures */

- ```
  swap (&address, register) {
      temp = M[address];
      M[address] = register;
      register = temp;
  }
  ```
  /* x86 */

- ```
  compare&swap (&address, reg1, reg2) {
      if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
  ```
  /* 68000 */


  ```

Implementing Locks with test&set

- Simple solution:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); // while busy
}
Release() {
   value = 0;
}
```

```
test&set (&address) {
  result = M[address];
  M[address] = 1;
  return result;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

Too Much Milk: Solution #2

- `Lock.Acquire()` – wait until lock is free, then grab
- `Lock.Release()` – unlock, waking up anyone waiting
- **atomic operations** – if two threads are waiting for the lock, only one succeeds to grab the lock

- Then, our milk problem is easy:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"

Shared Memory code for computing a sum

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

Thread 1

```
local_s1= 0
for i = 0, n/2-1
   local_s1 = local_s1 + sqr(A[i])

s = s + local_s1
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
   local_s2= local_s2 + sqr(A[i])

s = s +local_s2
```

- Since addition is associative, it's OK to rearrange order
  - Right?
- 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

Shared Memory code for computing a sum

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

Thread 1

```
local_s1= 0
for i = 0, n/2-1
    local_s1 = local_s1 + sqr(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 + sqr(A[i])
lock(lk);
s = s +local_s2
unlock(lk);
```

- Since addition is associative, it's OK to rearrange order
  - Right?
- 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

Performance Criteria for Synch. Ops

- Latency (time per op)
  - How long does it take if you always win
  - Especially when light contention
- Bandwidth (ops per sec)
  - Especially under high contention
  - How long does it take (averaged over threads) when many others are trying for it
- Traffic
  - How many events on shared resources (bus, crossbar,…)
- Storage
  - How much memory is required?
- Fairness
  - Can any one threads be “starved” and never get the lock?

Test-and-set based lock

Test-and-set instruction:

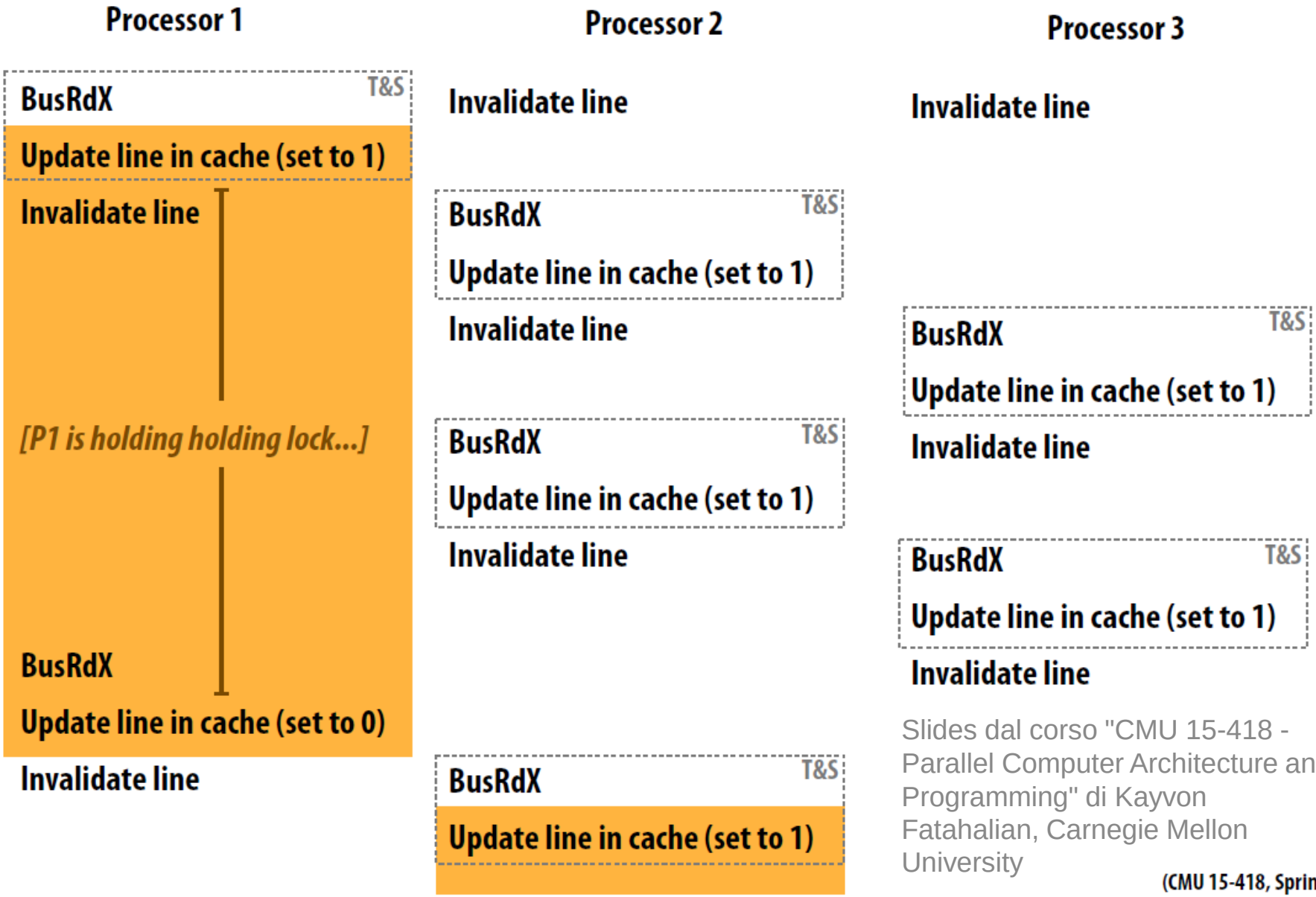
```
ts R0, mem[addr]        // atomically load mem[addr] into R0
                        // and set mem[addr] to 1
```

```
lock:       ts   R0, mem[addr]      // load word into R0
            bnz  R0, #0             // if 0, lock obtained

unlock:     st   mem[addr], #0      // store 0 to address
```

Slides dal corso "CMU 15-418 - Parallel Computer Architecture and Programming" di Kayvon Fatahalian, Carnegie Mellon University

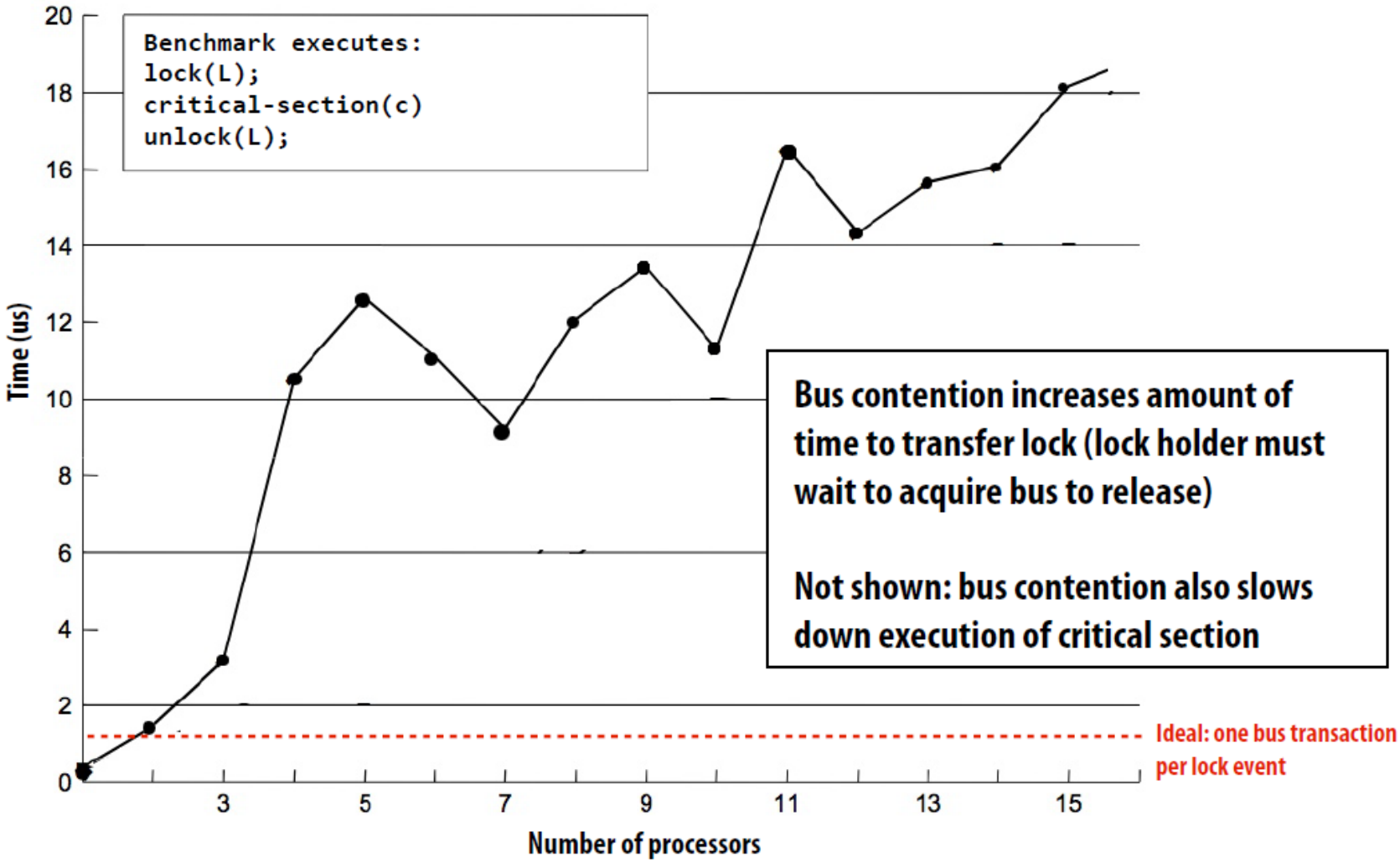
Test & set lock: consider coherence traffic



Slides dal corso "CMU 15-418 - Parallel Computer Architecture and Programming" di Kayvon Fatahalian, Carnegie Mellon University

Test-and-set lock performance

Benchmark: Total of N lock/unlock sequences (in aggregate) by P processors
Critical section time removed so graph plots only time acquiring/releasing the lock



Slides dal corso "CMU 15-418 - Parallel Computer Architecture and Programming" di Kayvon Fatahalian, Carnegie Mellon University

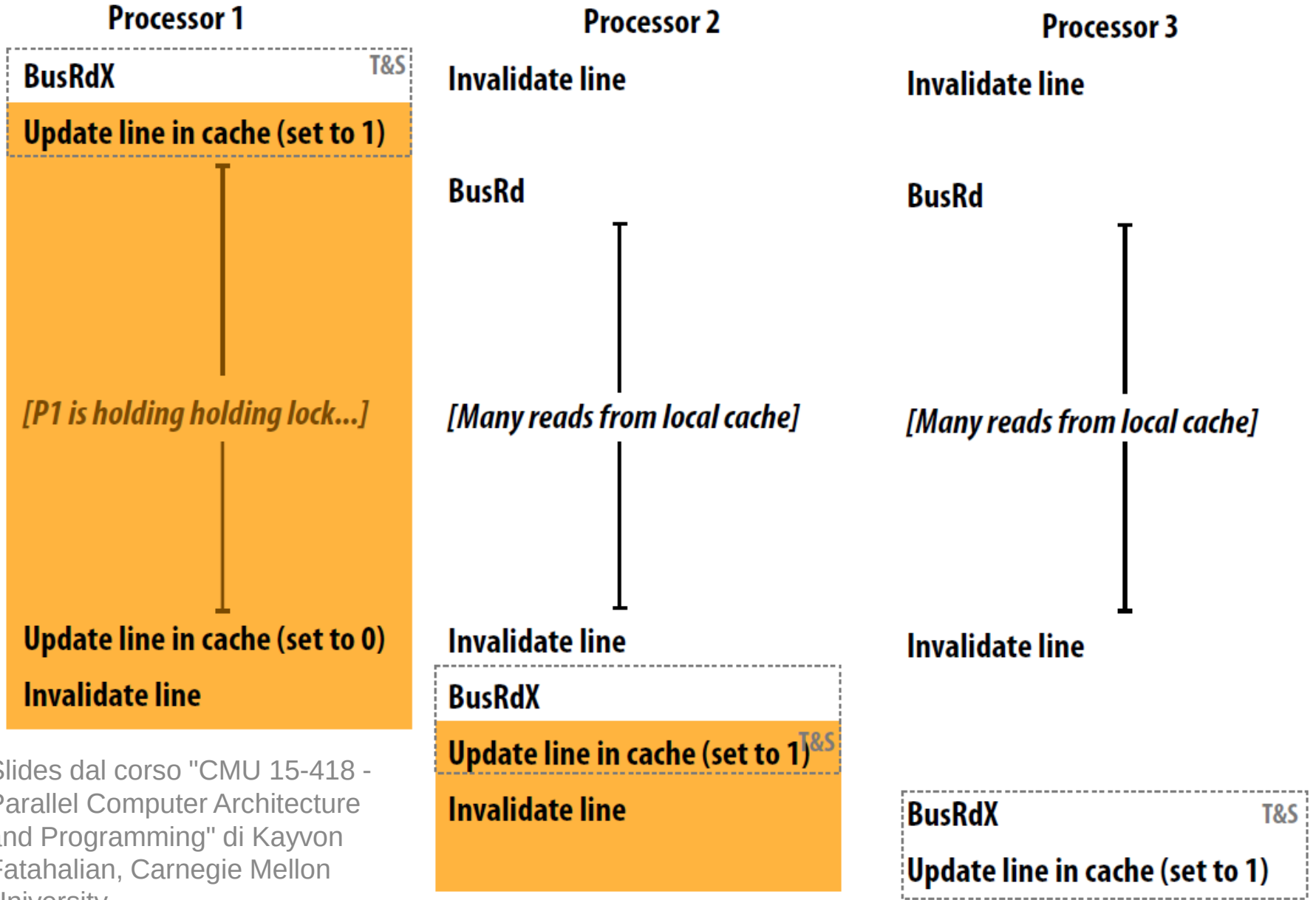
Test-and-test-and-set lock

```
void Lock(volatile int* lock) {
  while (1) {
    while (*lock != 0);       // while another processor has the lock
    if (test&set(*lock) == 0) // when lock is released, try to acquire it
      return;
  }
}

void Unlock(volatile int* lock) {
   *lock = 0;
}
```

Busy waiting no longer done with T&S on shared variable → reduces coherence traffic

Test & test & set lock: coherence traffic



Slides dal corso "CMU 15-418 - Parallel Computer Architecture and Programming" di Kayvon Fatahalian, Carnegie Mellon University

Test & test & set characteristics

- Higher latency than test & set in uncontended case
  - Must test... then test and set
- Generates much less bus traffic
  - One invalidation per waiting processor per lock release
- More scalable (due to less traffic)
- Storage cost unchanged
- Still no provisions for fairness

Test-and-set lock with backoff

Upon failure to acquire lock, delay for awhile before retrying

```
void Lock(volatile int* l) {
  int amount = 1;
  while (1) {
    if (test&set(*l) == 0)
      return;
    delay(amount);
    amount *= 2;
  }
}
```

Slides dal corso "CMU 15-418 - Parallel Computer Architecture and Programming" di Kayvon Fatahalian, Carnegie Mellon University

- Same uncontended latency as test and set
- Generates less traffic than test and set (not continually attempting to acquire lock)
- Improves scalability (due to less traffic)
- Storage cost unchanged
- Exponential backoff can cause severe unfairness
  - Newer requesters back off for shorter intervals

Ticket lock

Main problem with test & set style locks: upon release, all waiting processors attempt to acquire lock using test & set

```
struct lock {
   volatile int next_ticket;
   volatile int now_serving;
};

void Lock(lock* l) {
  int my_ticket = atomicIncrement(l->next_ticket);
  while (my_ticket != l->now_serving);
}

void unlock(lock* l) {
  l->now_serving++;
}
```

Slides dal corso "CMU 15-418 - Parallel Computer Architecture and Programming" di Kayvon Fatahalian, Carnegie Mellon University

Array-based lock

Each processor spins on a different memory address

Use fetch&op (below: atomicIncrement) to assign address on attempt to acquire

```
struct lock {
   volatile int status[P];
   volatile int head;
};

int my_element;

void Lock(lock* l) {
  my_element = atomicIncrement(l->head);  // assume circular inc
  while (l->status[my_element] == 1);
}

void unlock(lock* l) {
  l->status[next(my_element)] = 0;
}
```

O(1) traffic per release, but requires space linear in P

Three fundamental issues for shared memory multiprocessors

- **Coherence**
  Do I see the most recent data?

- **Synchronization**
  How to synchronize processes?
  - how to protect access to shared data?

- **Consistency**
  ***When** do I see a written value?*
  - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

Memory Consistency: The Problem

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

- Observation: If writes take effect immediately (are immediately seen by all processors), it is impossible that both if-statements evaluate to true

- But what if write invalidate is delayed …….
  - Should this be allowed, and if so, under what conditions?

Memory Consistency: The Problem

- **Cache-coherence is not enough!**
  - Many more subtle issues for parallel programs!
- **Memory Models**
  - Sequential consistency
  - Relaxed consistency models

Cache Coherence vs. Memory Model

- **Varying definitions!**

- **Cache coherence: a mechanism that propagates writes to other processors/caches of needed, recap:**
  - Writes are eventually visible to all processors
  - Writes to the same location are observed in order

- **Memory models: define the bounds on when the value is propagated to other processors**
  - E.g., sequential consistency requires *all* reads and writes to be ordered in program order

Memory Models

- **Need to define what it means to "read a location" and "to write a location" and the respective ordering!**
  - What values should be seen by a processor
- **First thought: extend the abstractions seen by a sequential processor:**
  - Compiler and hardware maintain data and control dependencies at all levels:

Two operations to the same location

```
Y=10
....
T = 14
Y=15
```

One operation controls execution of others

```
Y = 5
X = 5
T = 3
Y = 3
If (X==Y)
 Z = 5
....
```

Sequential Processor

- **Correctness condition:**
  - The result of the execution is the same as if the operations had been executed in the order specified by the program

    "program order"
  - A read returns the value last written to the same location

    "last" is determined by program order!
- **Consider only memory operations (e.g., a trace)**
- **N Processors**
  - P1, P2, …., PN
- **Operations**
  - Read, Write on shared variables (initial state: all 0)
- **Notation:**
  - P1: R(x):3 P1 reads x and observes the value 3
  - P2: W(x,5) P2 writes 5 to variable x

Terminology

- **Program order**
  - Deals with a *single* processor
  - Per-processor order of memory accesses, determined by program

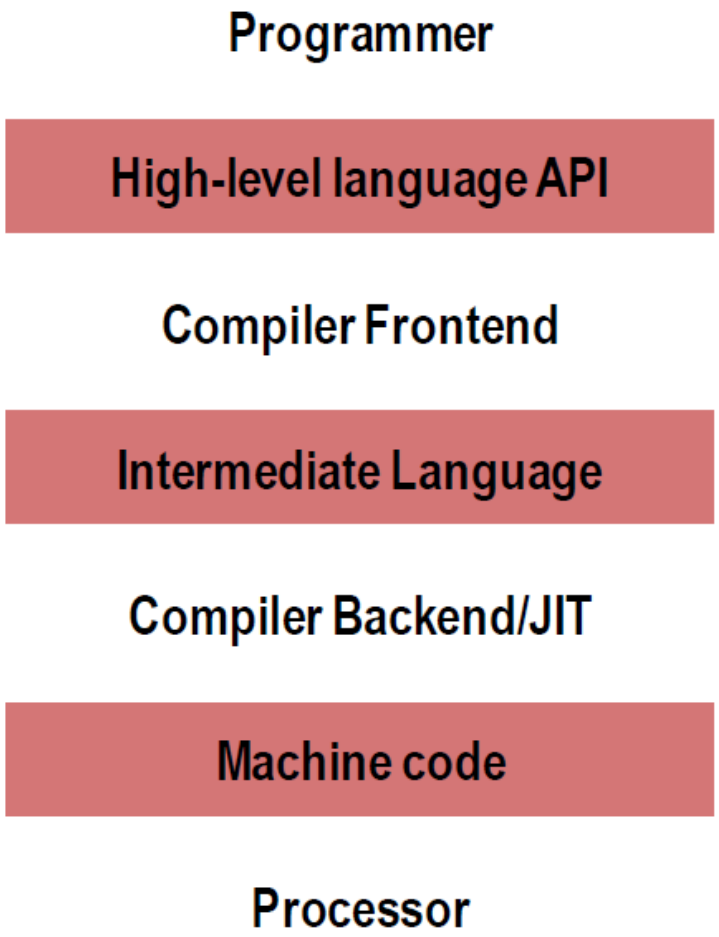
    Control flow
  - Often represented as trace

- **Visibility order**
  - Deals with operations on *all* processors
  - Order of memory accesses observed by one or more processors
  - E.g., "every read of a memory location returns the value that was written last"

    Defined by memory model

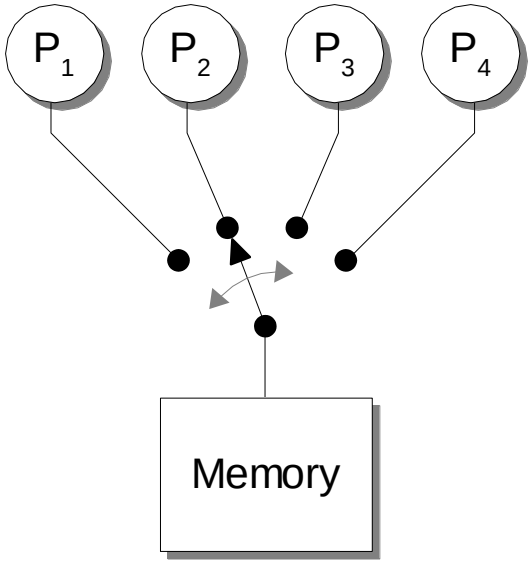
Memory Models

- **Contract at each level between programmer and processor**



Sequential Consistency

Lamport (1979): A multiprocessor is sequentially consistent if the result of any execution is the same as if the (memory) operations of all processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program



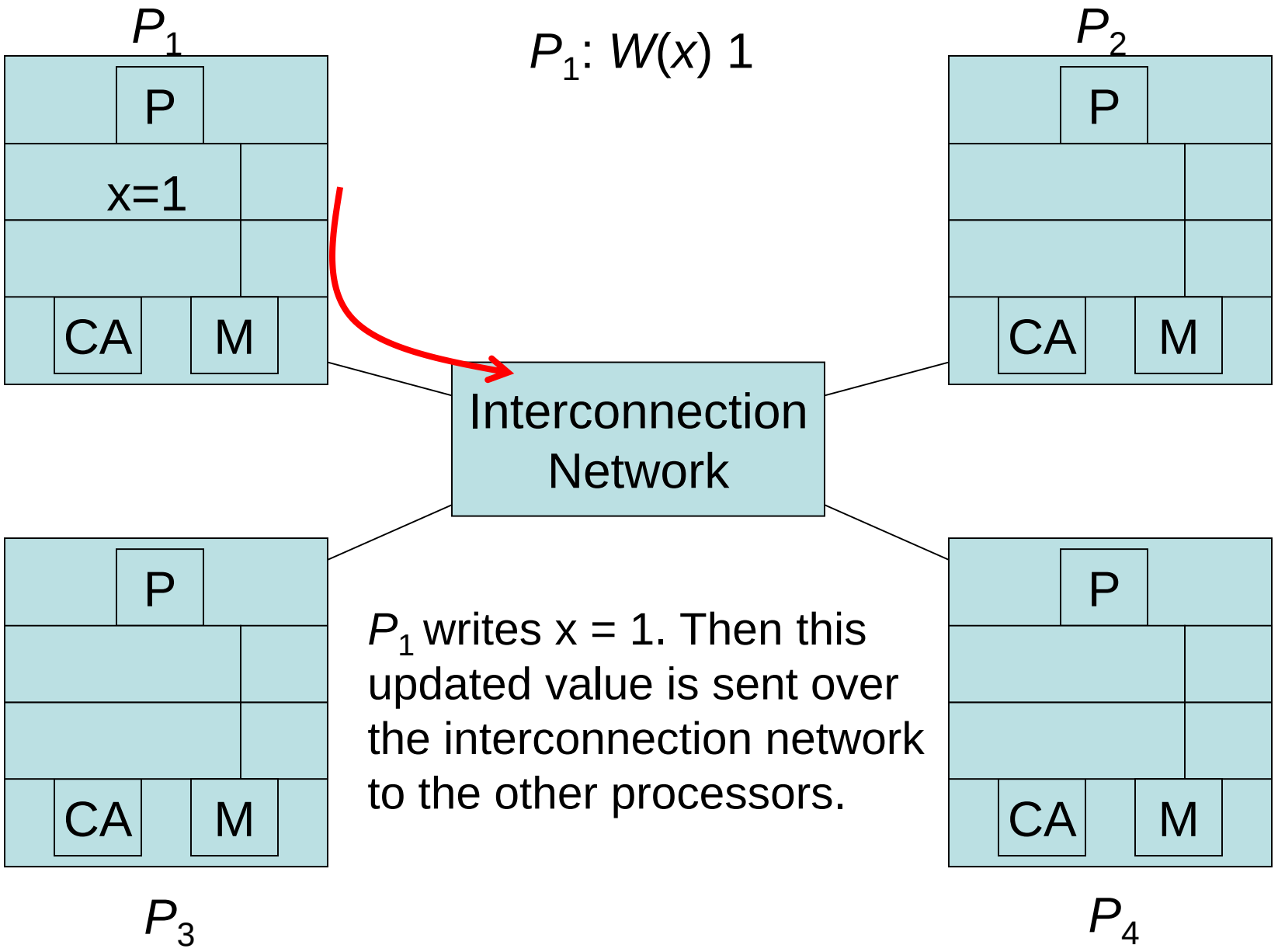
This means that all processors 'see' all loads and stores happening in the same order !!

Sequential Consistency Example

P_1:	W(x) 1		
P_2:		R(x) 0	R(x) 1
P_3:			
P_4:			

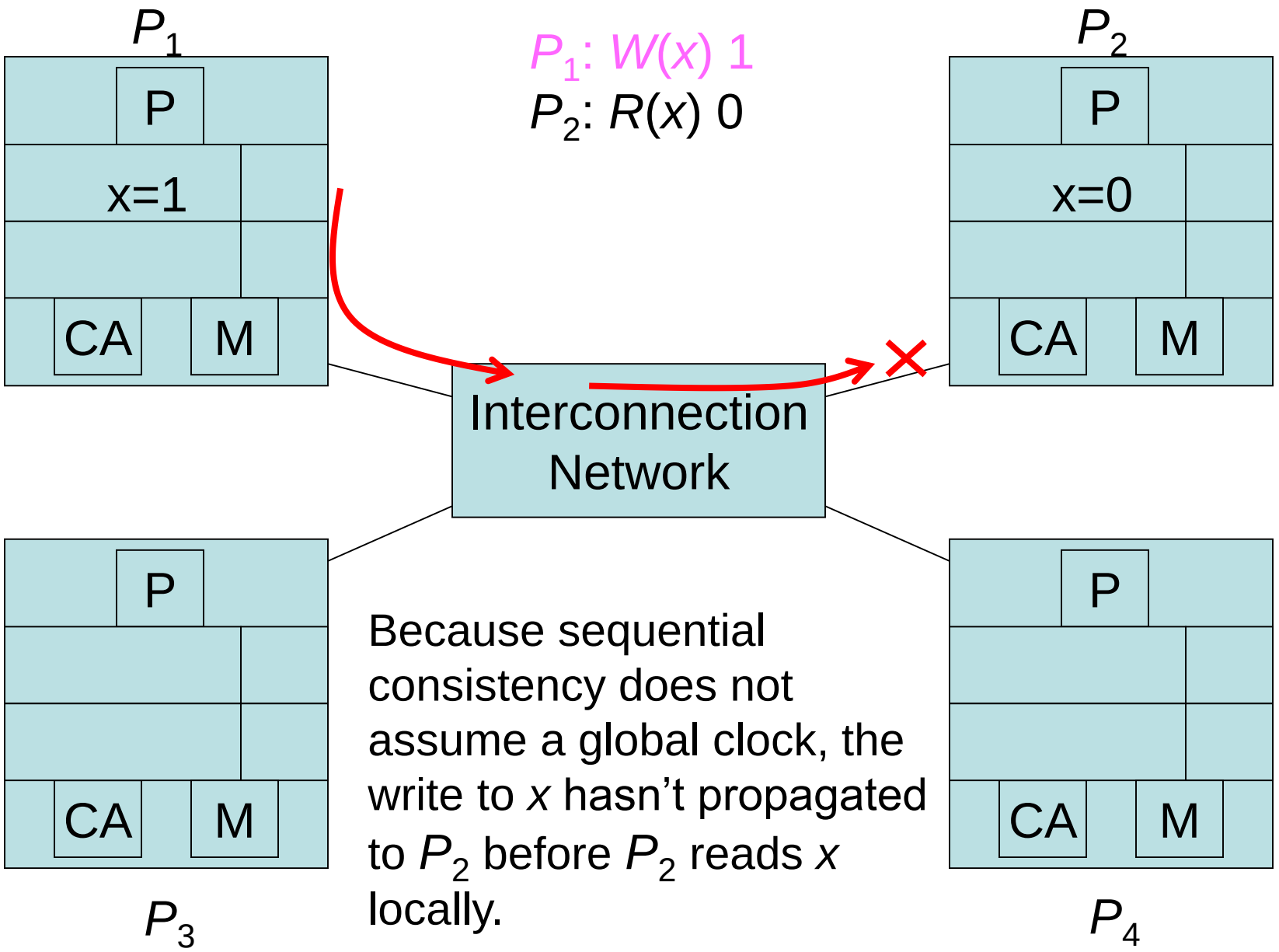
- P_2 does not see P_1's write of x = 1 before its first read of x, so it happens to have an out-of-date value.
- However, the write propagates to P_2 before its second read of x.
- This is legal under SC because:
  - Processors do not always have to see up-to-date values
  - Processors just need to see writes in the order they happen
- Note: it would also have been legal under SC for W(x) 1 to propagate to P_2 before its first read of x or after its second read of x.

Sequential Consistency Example (cont'd)



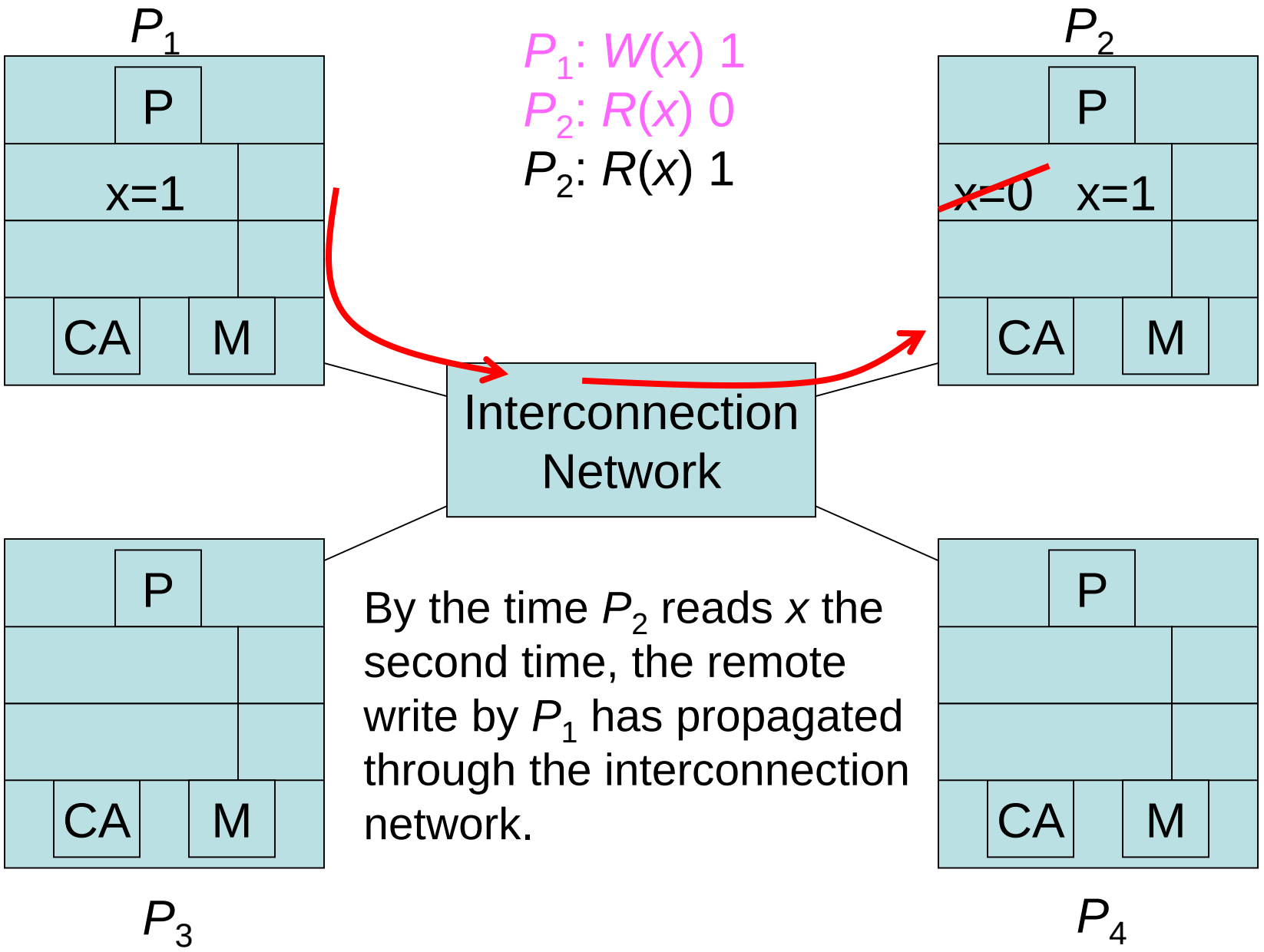
P_1 writes x = 1. Then this updated value is sent over the interconnection network to the other processors.

Sequential Consistency Example (cont'd)



Because sequential consistency does not assume a global clock, the write to *x* hasn't propagated to P_2 before P_2 reads *x* locally.

Sequential Consistency Example (cont'd)



How to implement Sequential Consistency

- Delay the completion of any memory access until all invalidations caused by that access are completed

- Delay next memory access until previous one is completed
  - delay the read of A and B (A==0 or B==0 in the example) until the write has finished (A=1 or B=1)

- Note: Under sequential consistency, we cannot place the (local) write in a write buffer and continue

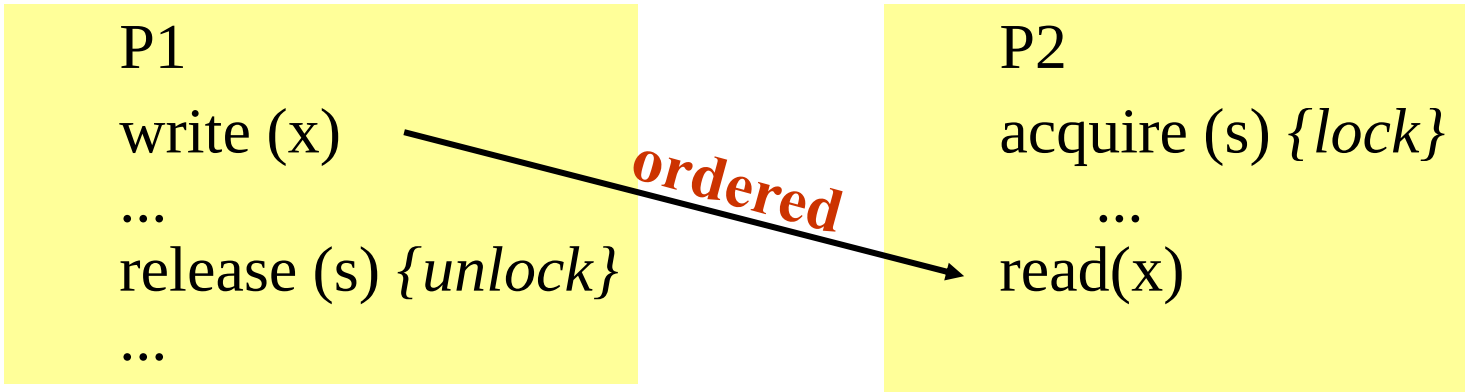
Cost of Sequential Consistency (SC)

- Enforcing SC can be quite expensive
  - Assume write miss = 40 cycles to get ownership,
  - 10 cycles = to issue an invalidate and
  - 50 cycles = to complete and get acknowledgement
  - Assume 4 processors share a cache block, how long does a write miss take for the writing processor if the processor is sequentially consistent?
- Waiting for invalidates : each write =sum of ownership time + time to complete invalidates
  - 10+10+10+10= 40 cycles to issue invalidate
  - 40 cycles to get ownership + 50 cycles to complete
  - =130 cycles ➔very long !

- Solutions:
  - Exploit latency-hiding techniques
  - Employ relaxed consistency

Sequential consistency overkill?

- Schemes for faster execution then sequential consistency
- Observation: Most programs are synchronized
  - A program is synchronized if all accesses to shared data are ordered by synchronization operations
- Example:



Relaxed Memory Consistency Models

- Key: ***(partially) allow reads and writes to complete out-of-order***
- Orderings that can be relaxed:
  - relax W $\Rightarrow$ R ordering
    - allows reads to bypass earlier writes (to different memory locations)
    - called ***processor consistency*** or ***total store ordering***
  - relax W $\Rightarrow$ W
    - allow writes to bypass earlier writes
    - called ***partial store order***
  - relax R $\Rightarrow$ W *and* R $\Rightarrow$ R
    - ***weak ordering***, ***release consistency***, Alpha, PowerPC
- Note, seq. consistency means:
  - W $\Rightarrow$ R, W $\Rightarrow$ W, R $\Rightarrow$ W and R $\Rightarrow$ R

Relaxed consistency model: Weak consistency

- Programmer specifies regions within which global memory operations can be reordered
- Processor has **fence instruction**:
  - all data operations before fence in program order must complete before fence is executed
  - all data operations after fence in program order must wait for fence to complete
  - fences are performed in program order
- Implementation of fence:
  - processor has counter that is incremented when data op is issued, and decremented when data op is completed
- Example: PowerPC has SYNC instruction
- Language constructs:
  - OpenMP: flush
  - All synchronization operations like lock and unlock act like a fence

Weak ordering picture