CMPEN 431
Introduction to Computer Architecture
Fall 2022

Multiprocessor (1): Cache Coherence (2) + Consistency

Kiwan Maeng

Slides adapted from ECE6100 @ GeorgiaTech
by Hsien-Hsin Sean Lee &
Computer Organization and Design, 5th Edition,
Patterson & Hennessy, © 2014, MK
Part of the slides adapted from UTAustin's James Bornholt's Blog

Project 1 (Due 11/10), Quiz (Due 11/14)

Implication on Multi-Level Caches

- How to guarantee coherence in a multi-level cache hierarchy
- Snoop all cache levels?

- Maintaining *inclusion property*
- Ensure data in the outer level must be present in the inner level
- Only snoop the outermost level (e.g. L2)
- L2 needs to know L1 has write hits
- Use Write-Through cache
- Use Write-back but maintain another "modified-but-stale" bit in L2

Inclusion Property

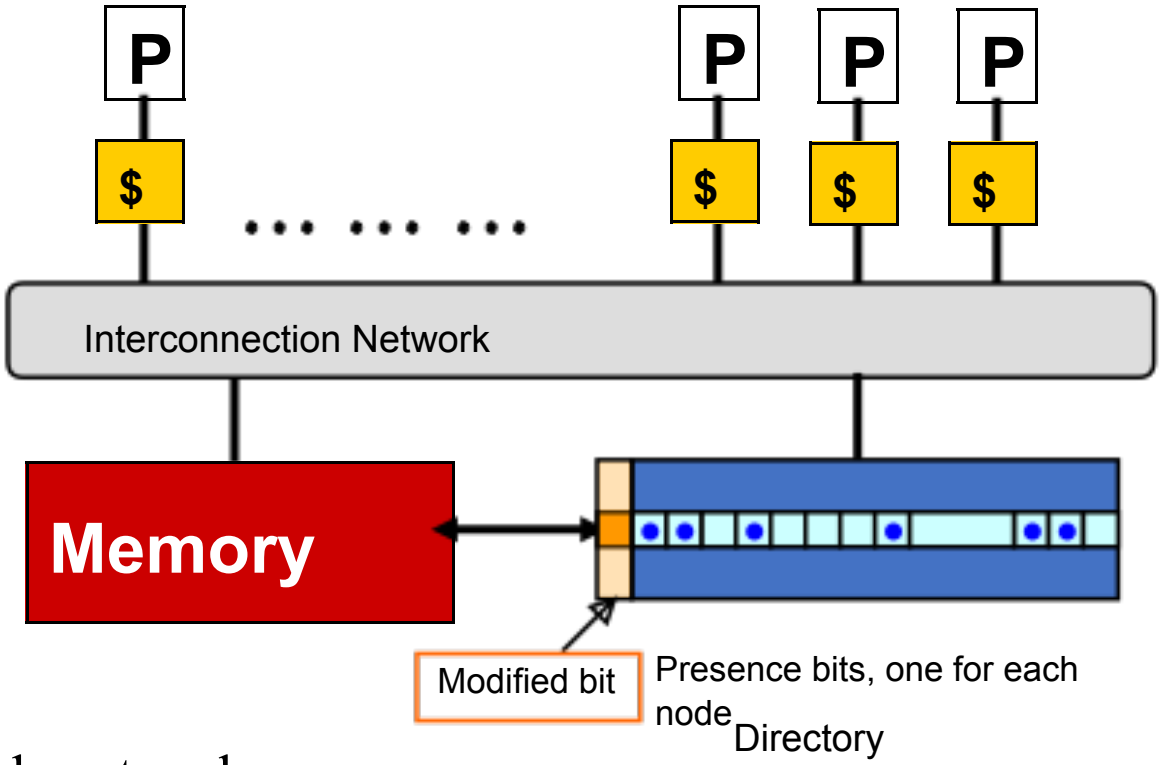
- Not so easy …
- Replacement: Different bus observes different access activities, e.g. L2 may replace a line frequently accessed in L1
- Split L1 caches: Imagine all caches are direct-mapped.
- Different cache line sizes

Inclusion Property

- Use specific cache configurations
- E.g., DM L1 + bigger DM or set-associative L2 with the same cache line size

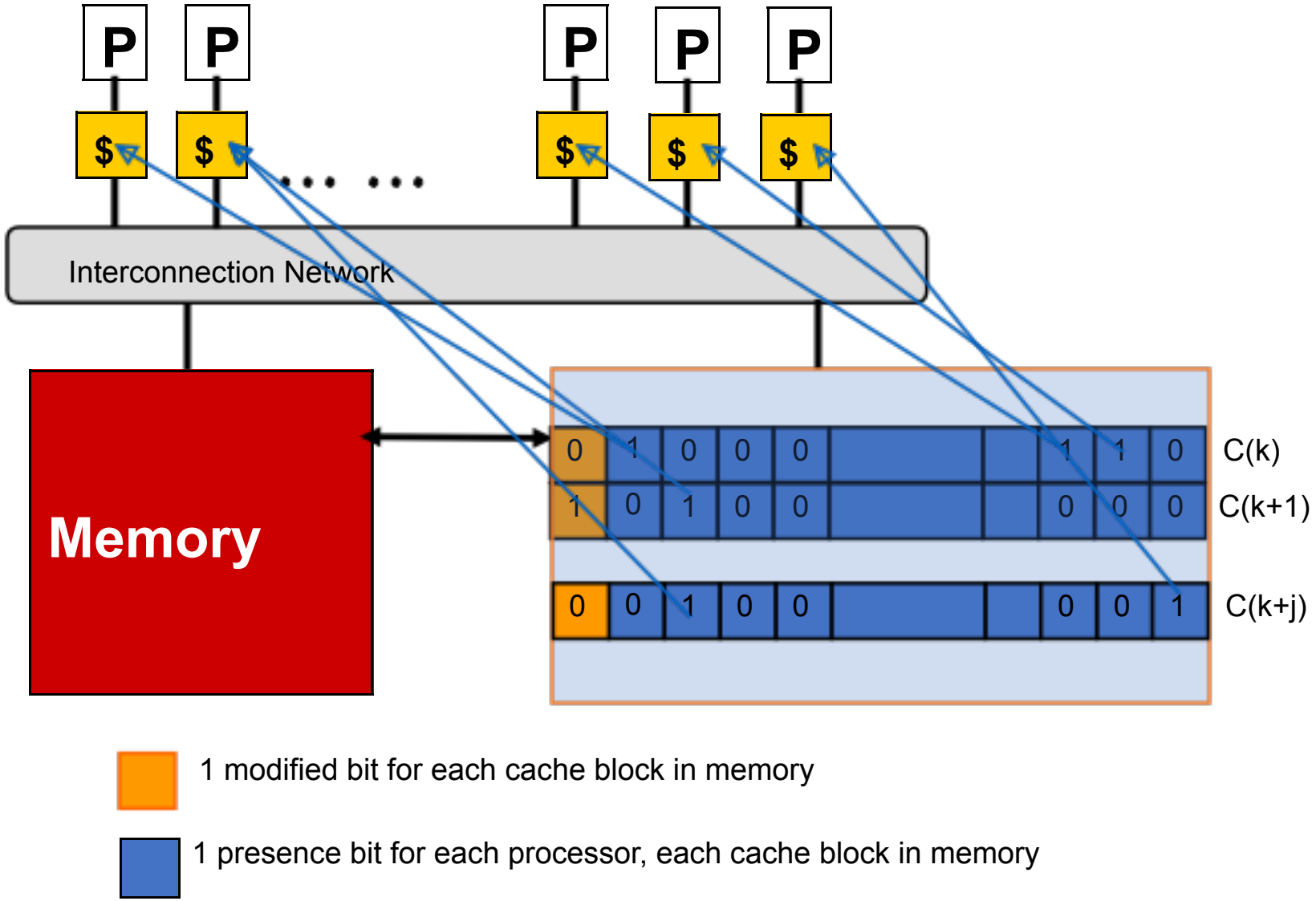
- Explicitly propagate L2 action to L1
- L2 replacement will flush the corresponding L1 line
- Observed BusRdX bus transaction will invalidate the corresponding L1 line
- To avoid excess traffic, L2 maintains *an Inclusion bit* for filtering (to indicate in L1 or not)

Directory-based Coherence Protocol

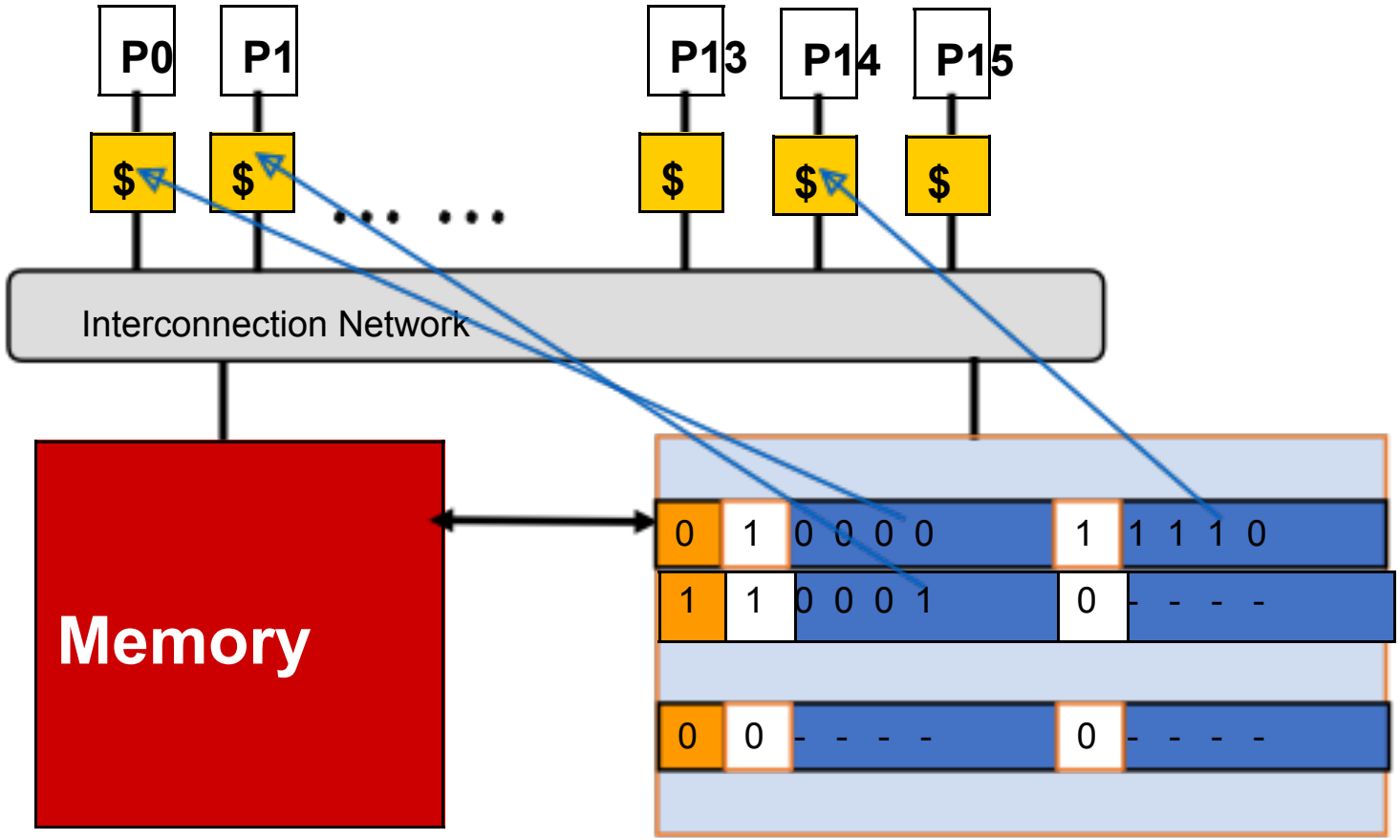


- Snooping-based protocol
- Broadcast to every processor (but you only need to tell the sharers)
- Only one message can go through the bus each time
- Not scalable large shared memory systems
- Directory protocol
- Directory-based control of who has what;
- HW overheads to keep the directory (~ # lines * # processors)
- Directory can be multi-banked, distributed, cached, etc…

Directory-based Coherence Protocol



Directory-based Coherence Protocol (Limited Dir)

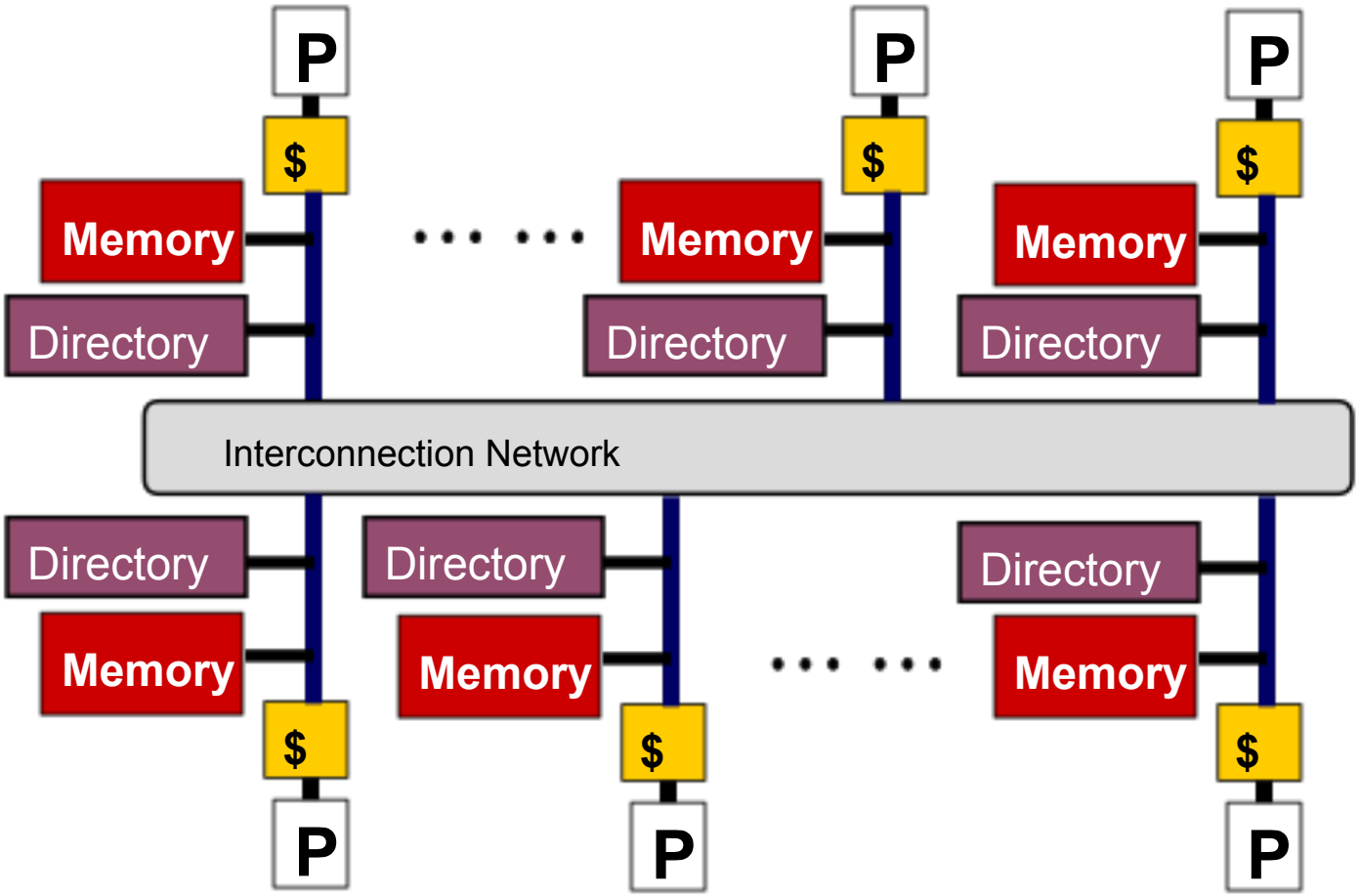


1 modified bit for each cache block in memory

Presence encoding is NULL or not

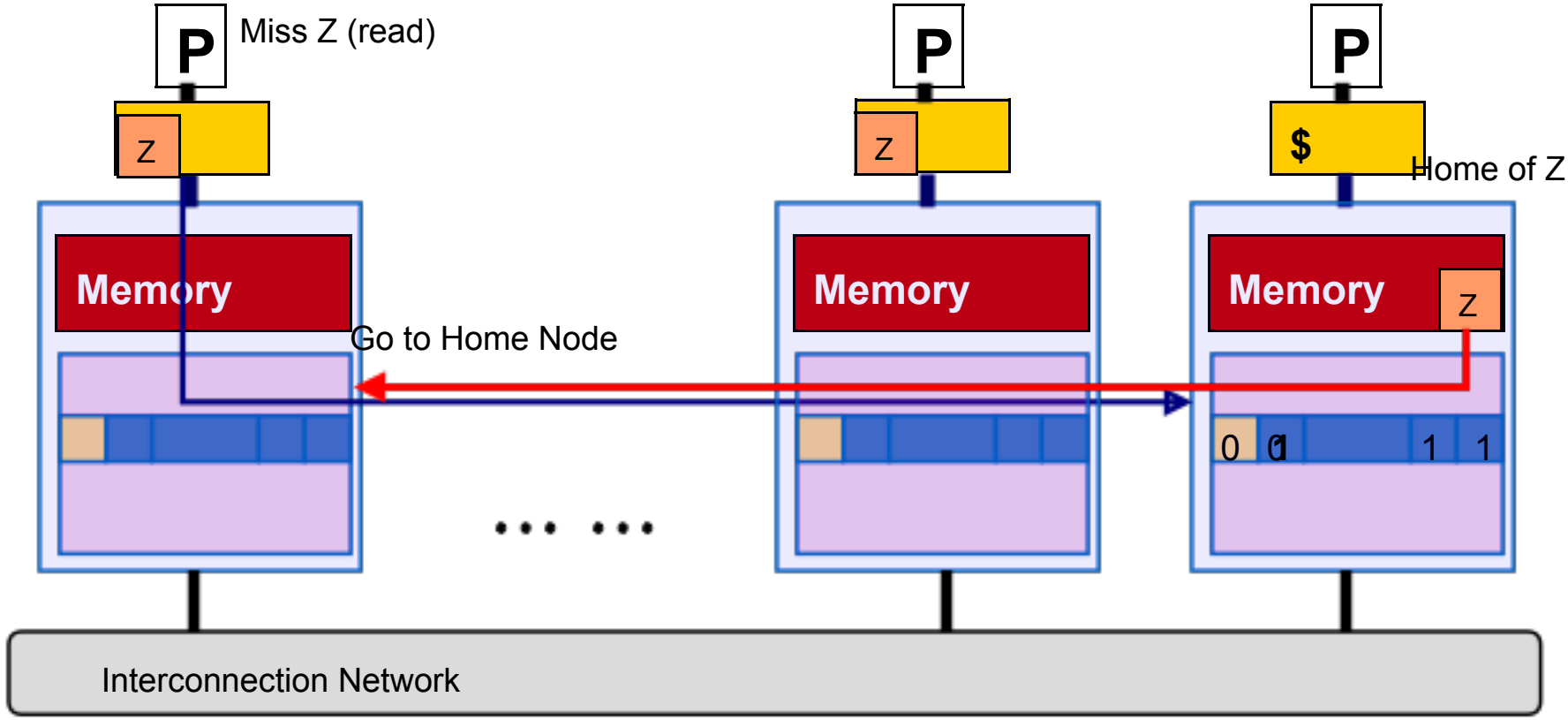
Encoded Present bits (lg2N),
each cache line can reside in **2 processors in this example**

Distributed Directory Coherence Protocol



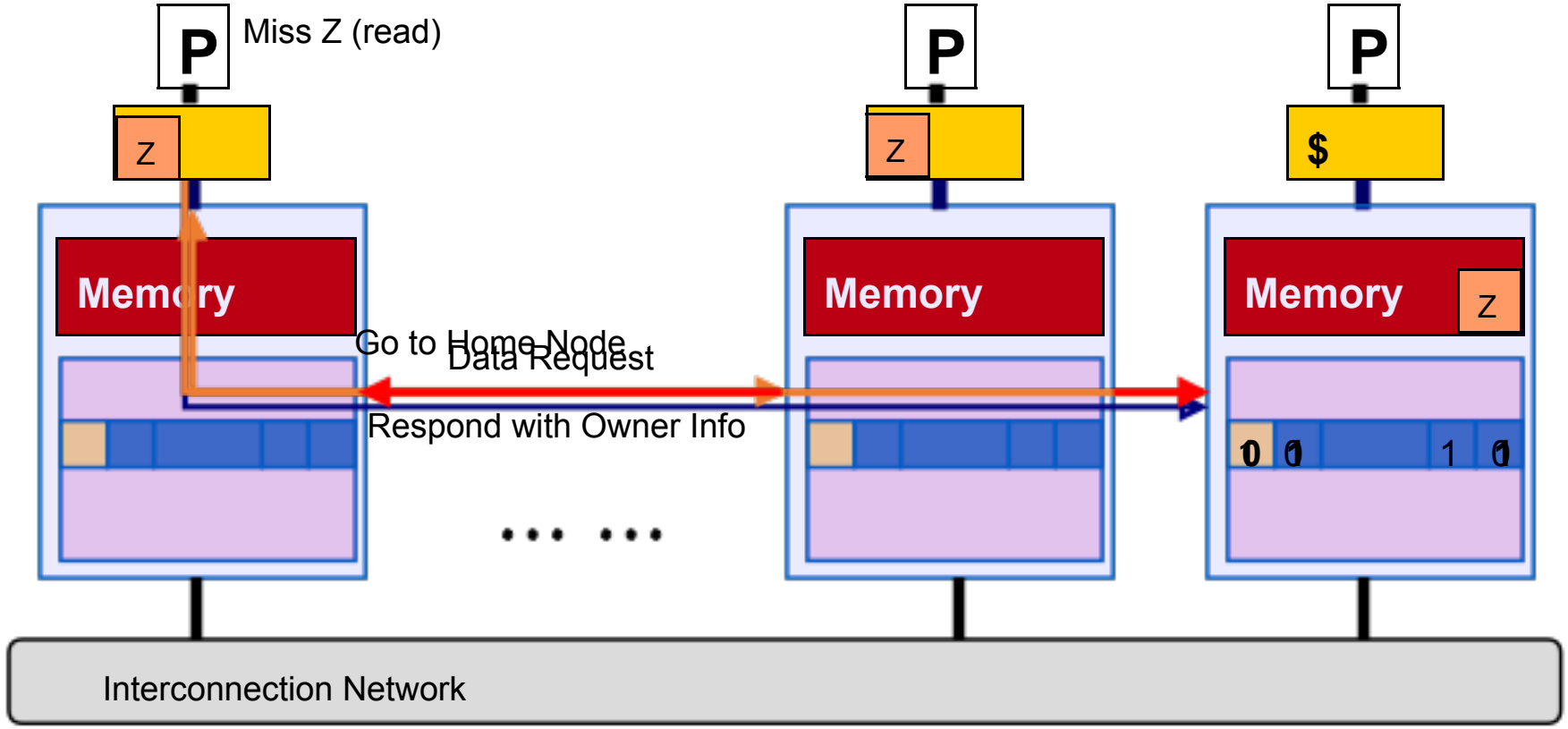
- Centralized directory is less scalable (contention)
- Distributed shared memory (DSM) for a large MP system
- Interconnection network is no longer a shared bus
- Maintain cache coherence (CC-NUMA)
- Each address has a “home”

Directory Coherence Protocol: Read Miss

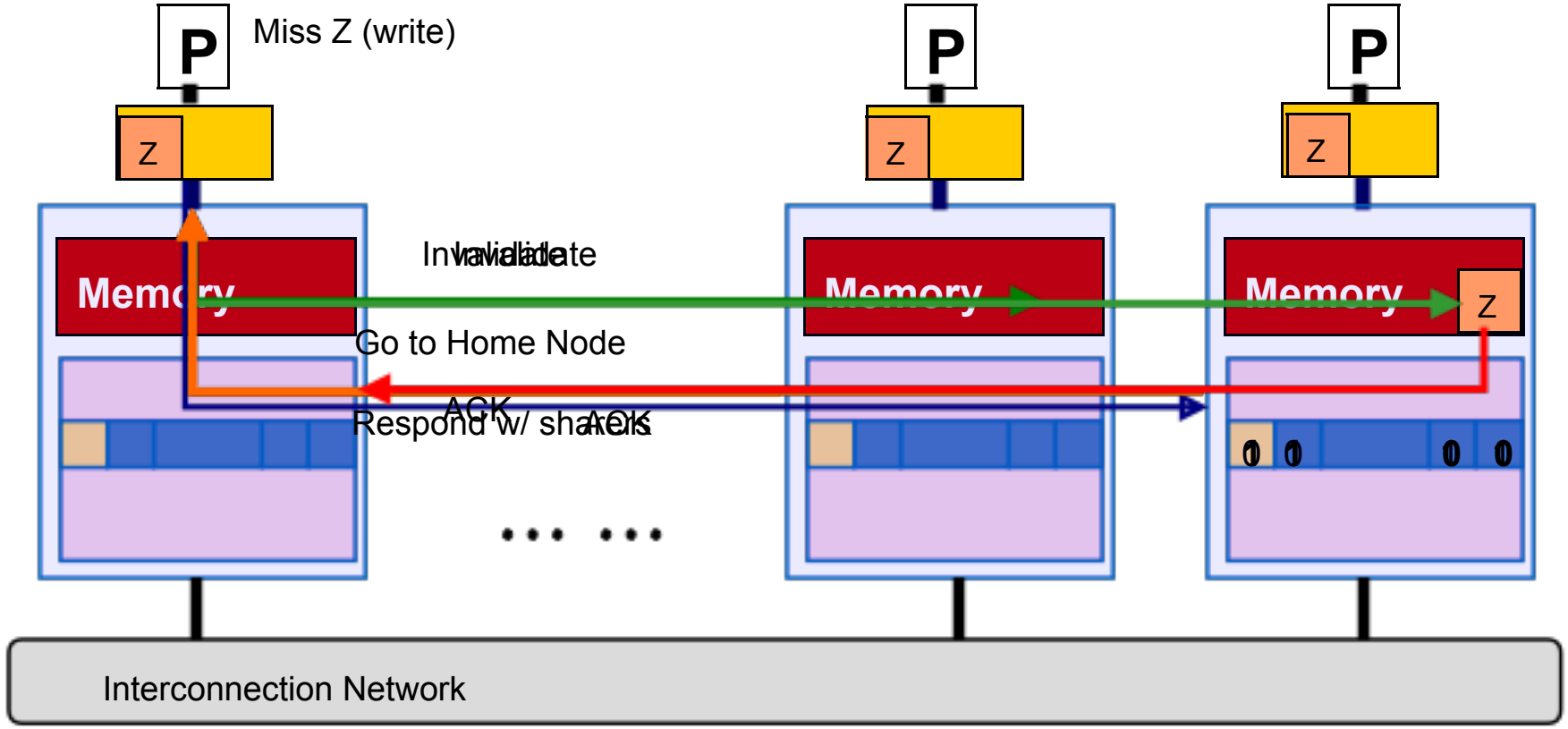


Data Z is shared (clean)

Directory Coherence Protocol: Read Miss



Directory Coherence Protocol: Write Miss



Write Z can proceed in P0

Memory Consistency Issue

- What do you expect for the following codes?

Initial values
A=0
B=0

Thread 1

```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```

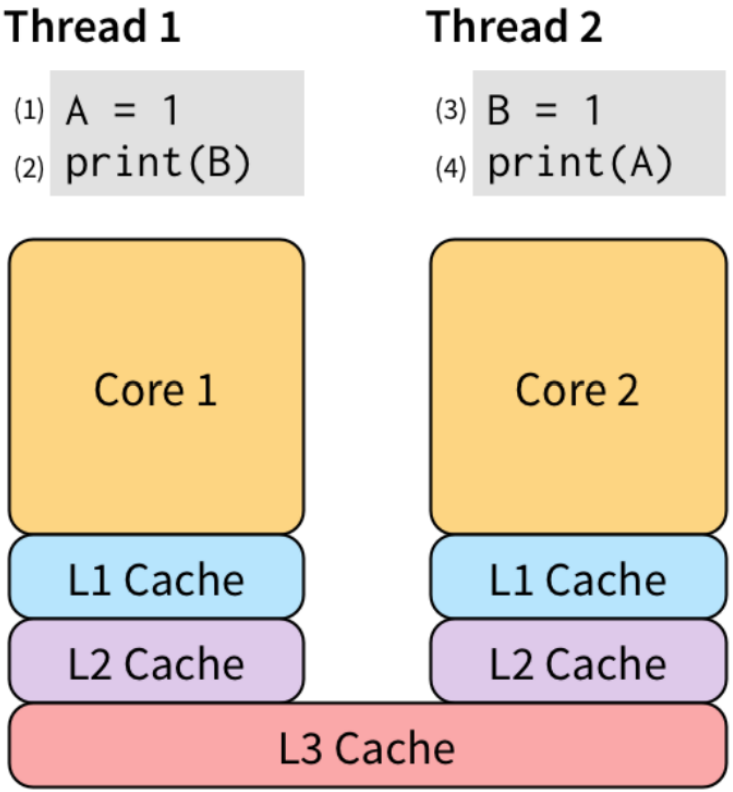
Is it possible to print A=0, B=0?

Memory Consistency Model

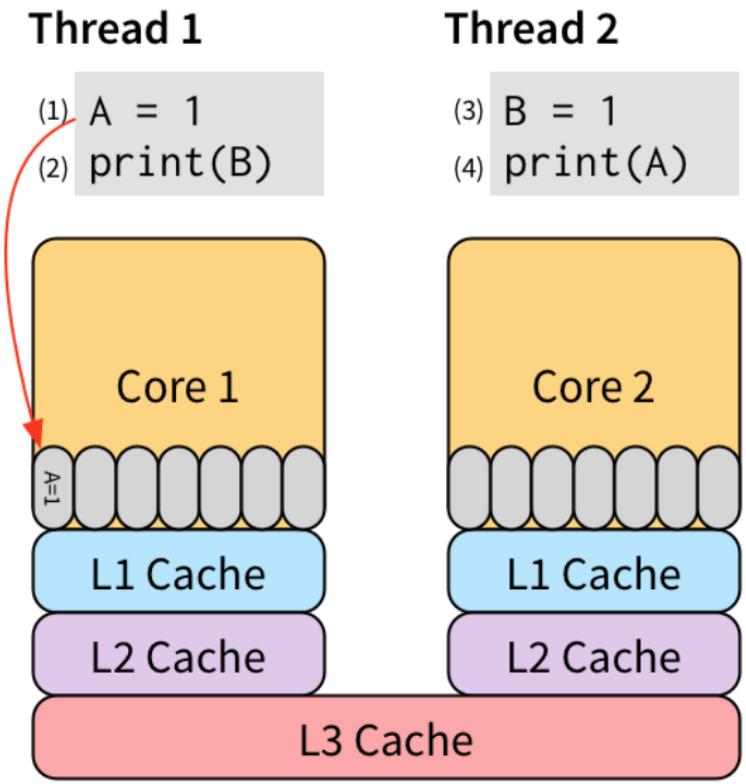
- Programmers anticipate certain memory ordering and program behavior
- The intuitive ordering may be different with the actual ordering that can be observed because of hardware specifics. However, note that we are NOT talking about,
- Compiler reordering instructions, or
- OoO processor reordering instructions inside the reservation station

- A memory consistency model specifies the legal ordering of memory events when several processors access the shared memory locations

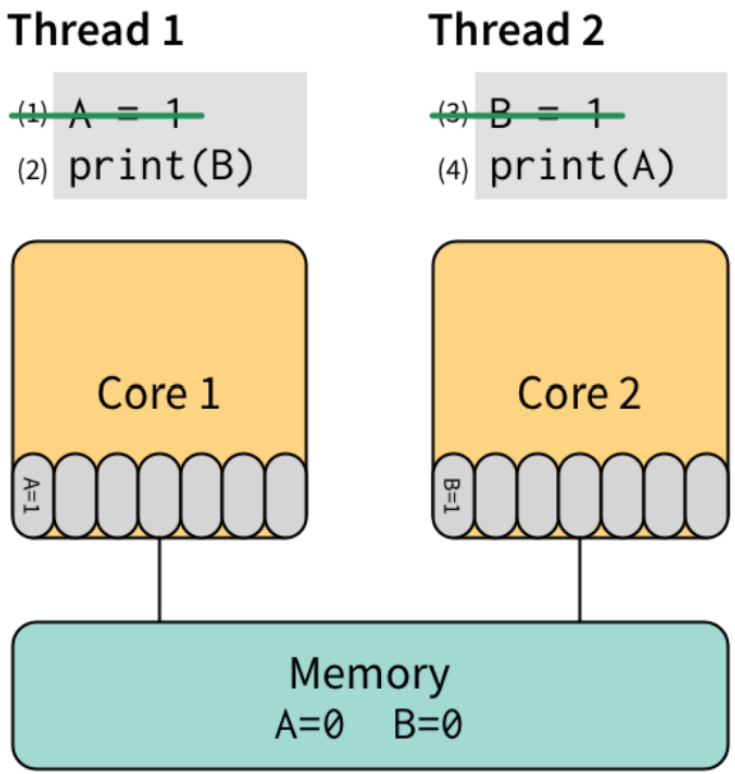
Why Can the Memory Possible Be Reordered?



Why Can the Memory Possible Be Reordered?



Why Can the Memory Possible Be Reordered?



Shouldn't this be prohibited?

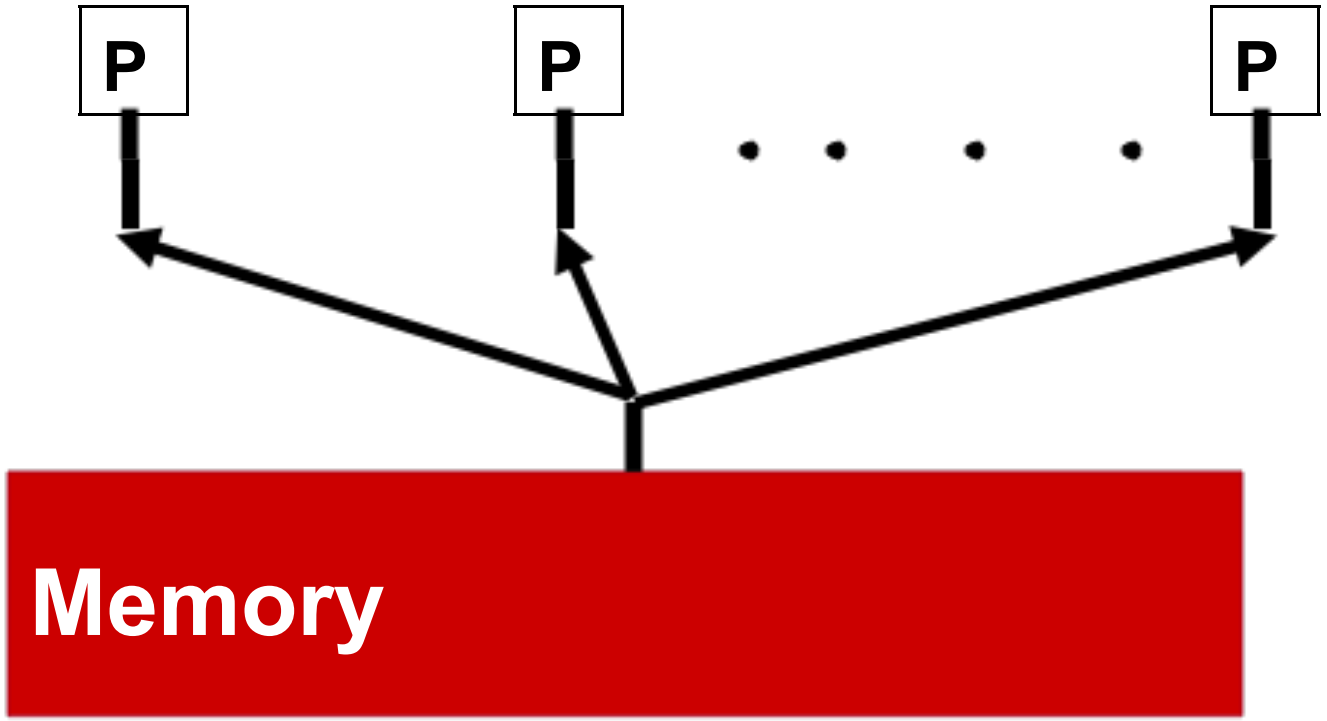
We can prohibit it

But if we allow it, your computer will be faster

So how about allowing it, and be mindful about what can happen while programming?

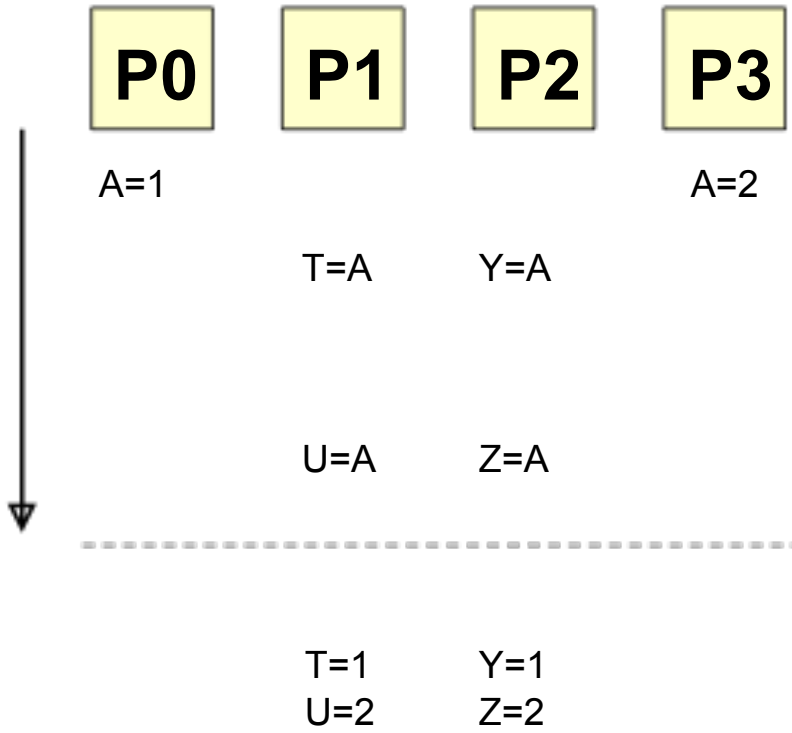
What is allowed and what is not allowed? => Memory consistency model

Sequential Consistency (SC) [Leslie Lamport]

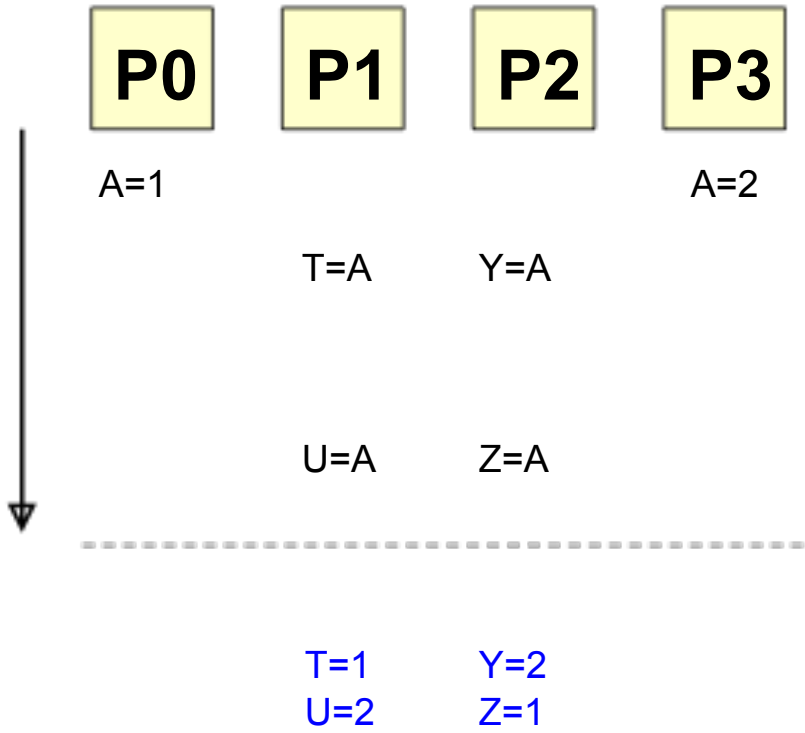


- An MP is Sequentially Consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.
- Two properties
- Program ordering
- Write atomicity (All writes to any location should appear to all processors in the same order)
- Intuitive to programmers

SC Example



Sequentially Consistent



Violating Sequential Consistency!
(but possible for processor consistency)

Maintain Program Ordering (SC)

Flag1 = Flag2 = 0

P1

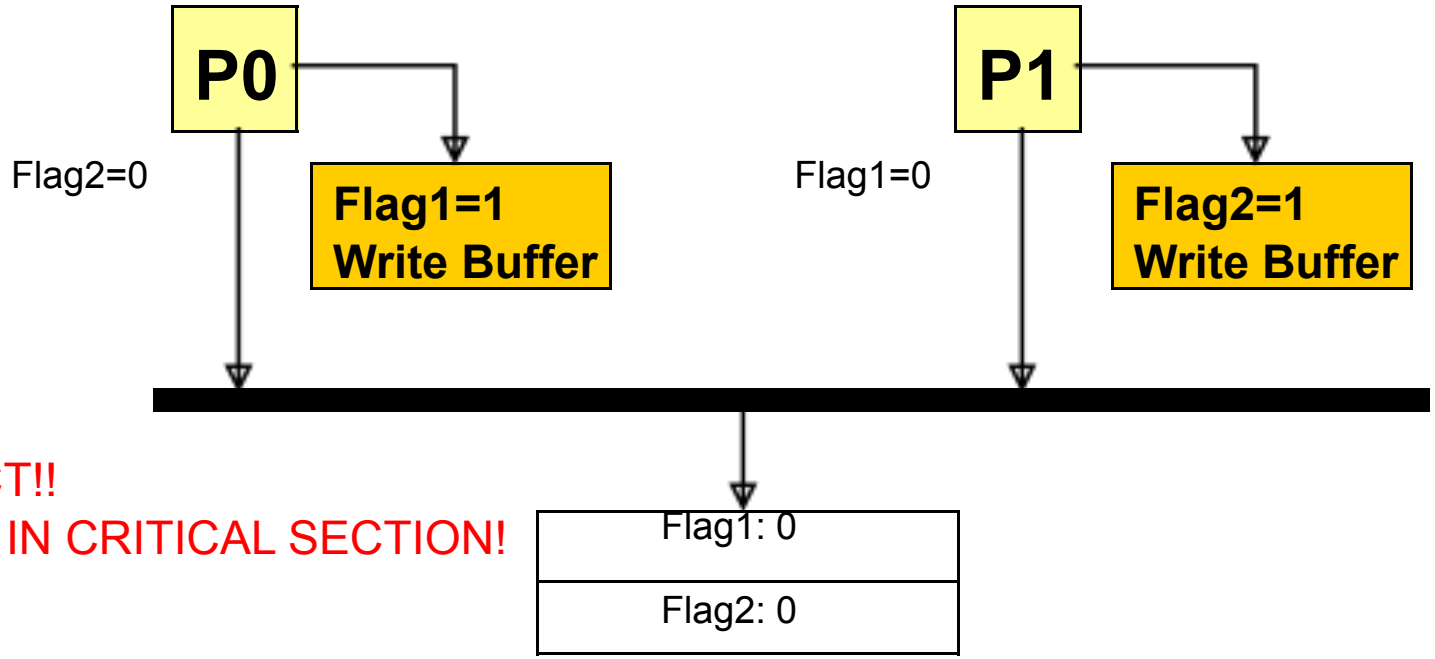
```
Flag1 = 1
if (Flag2 == 0)
 enter Critical Section
```

P2

```
Flag2 = 1
if (Flag1 == 0)
 enter Critical Section
```

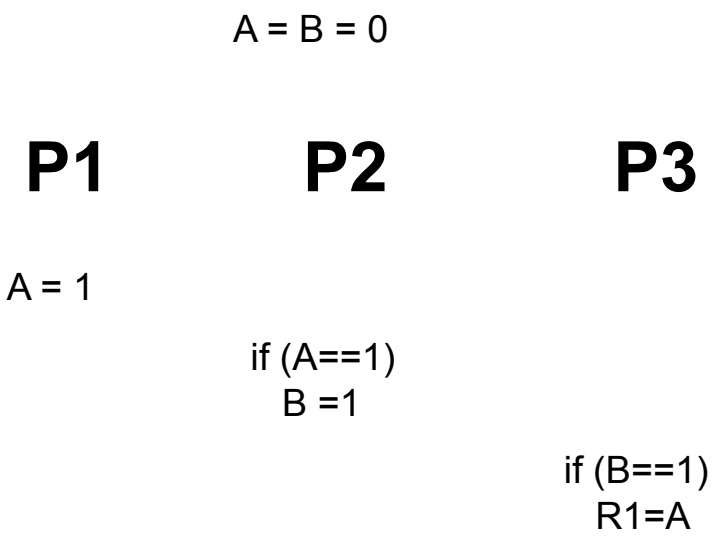
- Dekker's algorithm
- Only one processor is allowed to enter the CS

Caveat: implementation fine with uni-processor,but violate the ordering of the above



INCORRECT!!
BOTH ARE IN CRITICAL SECTION!

Atomic and Instantaneous Update (SC)



- Update (of A) must take place atomically to all processors
- A read cannot return the value of another processor's write until the write is made visible by "all" processors

Atomic and Instantaneous Update (SC)

A = B = 0

P1 **P2** **P3**

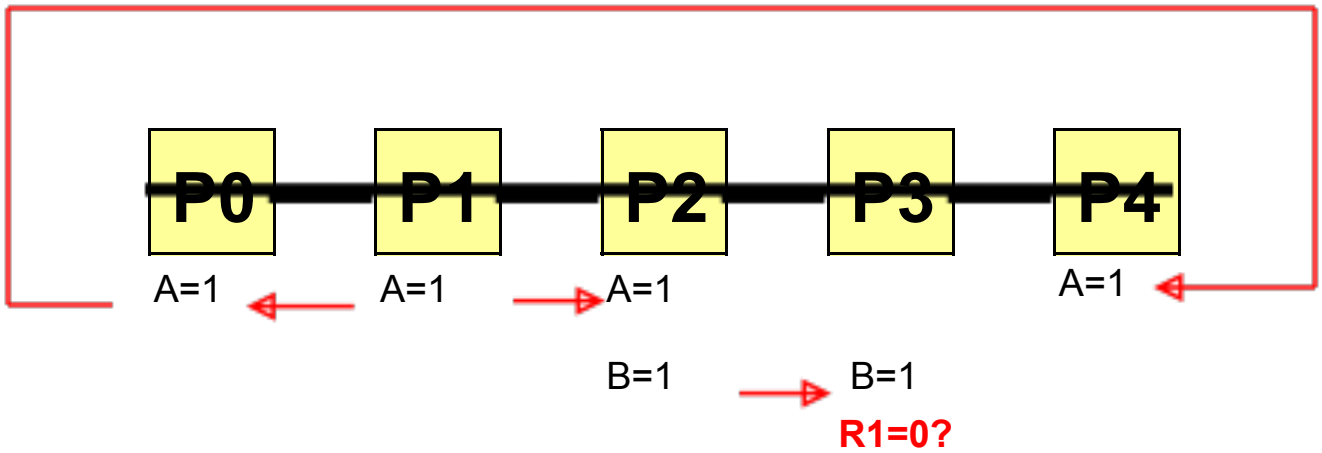
A = 1

if (A==1)
B =1

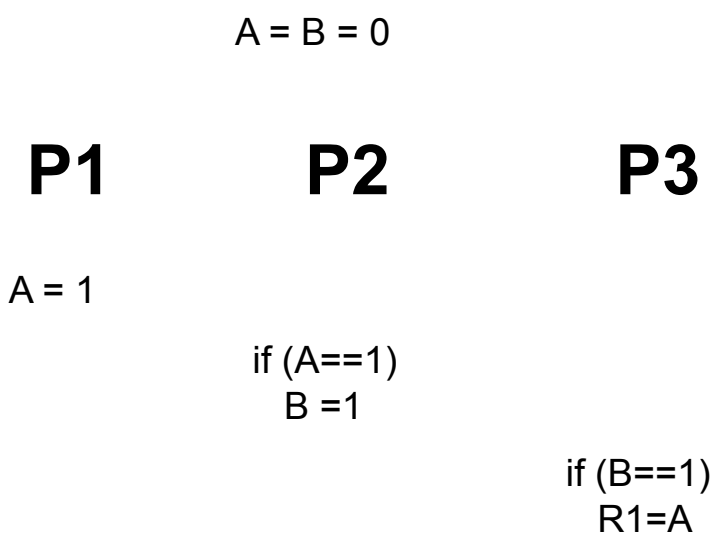
if (B==1)
R1=A

- Update (of A) must take place atomically to all processors
- A read cannot return the value of another processor's write until the write is made visible by "all" processors

Caveat when an update is not atomic to all …



Atomic and Instantaneous Update (SC)



- Caches also make things complicated
- P3 caches A and B
- A=1 will not show up in P3 until P3 reads it in R1=A

Relaxed Memory Models

- How to relax program order requirement?
- Load bypass store
- Load bypass load
- Store bypass store
- Store bypass load

- How to relax write atomicity requirement?
- Read others' write early
- Read own write early

Relaxed Consistency

- Processor Consistency
- Used in P6
- Write visibility could be in different orders of different processors (not guarantee write atomicity)
- Allow loads to bypass independent stores in each individual processor
- To achieve SC, explicit synchronization operations need to be substituted or inserted
- Read-modify-write instructions
- Memory fence instructions

Processor Consistency

A = Flag = 0

P1	P2
A = 1	while (Flag==0);
Flag=1	Print A

Intuitive for event synchronization

“A” must be printed “1”

Processor Consistency

Flag1 = Flag2 = 0

P1

```
Flag1 = 1
if (Flag2 == 0)
 enter Critical Section
```

P2

```
Flag2 = 1
if (Flag1 == 0)
 enter Critical Section
```

- Allow load bypassing store to a different address
- Unlike SC, cannot guarantee mutual exclusion in the critical section

Processor Consistency

Flag1 = Flag2 = 0

P1

```
Flag1 = 1
// memory barrier here
if (Flag2 == 0)
 enter Critical Section
```

P2

```
Flag2 = 1
// memory barrier here
if (Flag1 == 0)
 enter Critical Section
```

- Allow load bypassing store to a different address
- Unlike SC, cannot guarantee mutual exclusion in the critical section
- Memory barrier (fence) is needed!

Processor Consistency

A = B = 0

P1	P2	P3
A = 1		
	if (A==1) B =1	
		if (B==1) R1=A

B=1;R1=0 is a possible outcome

Since PC allows A=1 to be visible in P2 prior to P3

Compilers Reorder Too!

- We focused on what hardware do. However, compilers also reorder code all the time and similar problems may happen!
- Languages has their own memory model to define what is allowed or not allowed.
- You can put a “compiler barrier” to prevent reordering, similar to memory barrier.

Synchronization

- Need to be able to coordinate processes working on a common task; sharing data must be coordinated carefully.
- We already looked at some:
- Critical section: only one thread can enter the section
- Lock/mutex: An abstract concept you have to grab to access a shared resource – only one thread can grab a lock/mutex
- Semaphore: N number of threads can grab at the same time
- To achieve:
- mutual exclusion – restrict data access to one core at a time
- sequential ordering – must complete the first operation before the second operation can begin

1. Need an architecture-supported arbitration mechanism to decide which core gets access to the lock variable

- Single bus provides an arbitration mechanism, since the bus is the only path to memory – the core that gets the bus wins

2. Need an architecture-supported operation that locks the variable

- E.g., atomic exchange, load-linked and store-conditional

Why Do We Need Synchronization?: Example

```
void deposit(int amount) {
   balance += amount;
}
```

```
lw $t0, 0($s0)   # $s0 holds the address of balance
add $t0, $t0, $s1   # $s1 holds amount
sw $t0, 0($s0)
```

Why Do We Need Synchronization?: Example

```
Thread 1
deposit(100);
```

```
lw $t0, 0($s0)
add $t0, $t0, $s1
sw $t0, 0($s0)
```

```
Thread 1
deposit(200);
```

```
lw $t0, 0($s0)
add $t0, $t0, $s1
sw $t0, 0($s0)
```

Why Do We Need Synchronization?: Example

```
 Thread 1
 deposit(100);

lw $t0, 0($s0)


add $t0, $t0, $s1
sw $t0, 0($s0)
```

```
 Thread 1
 deposit(200);


lw $t0, 0($s0)


add $t0, $t0, $s1
sw $t0, 0($s0)
```

Data race!

Need For a Lock!

```
void deposit(int amount) {
    acquire_lock(L);
    balance += amount;
    release_lock(L);
}
```

Implementing a Lock

```
// Initially L is zero
int acquire_lock(int &L) {
   while (L == 1) {}
   L = 1;
}
void release_lock(int &L) {
   L = 0;
}
```

Will this work??

Implementing a Lock (Correctly!): Atomic Exchange

- Atomic exchange: atomically swap the values of the register file and the memory (in one indivisible instruction!)

```
        $t0 = 1;
Lock: EXCH $t0, 0($s0)   # $s0 is the address for L
        bne $t0, $zero, Lock
```

- Incur a lot of memory writes
- Does both read and write – cannot be done in a single MEM stage
- MIPS do not have an EXCH instruction

Implementing a Lock (Correctly!): LL and SC

- Load-linked (LL): Acts like a normal load
- store-conditional (SC): Only successfully stores if nobody has touched the memory location since the last LL to the location. If succeed, the register value becomes 1. Otherwise, the register value becomes 0.

```
Lock: ll $t0, 0($s0)   # $s0 is the address for L
      bne $t0, $zero, Lock
      addi $t0, $t0, 1
      sc $t0, 0($s0)
      beq $t0, $zero, Lock
```

- SC does not store if it fails – does not incur memory traffic
- Can be implemented easily on top of coherence (e.g., remember bus snooping?)

Directly Implementing a Critical Section with LL&SC

- We can even directly implement a small critical section from before.

```
Retry: ll $t0, 0($s0)   # $s0 is the address for balance
       addi $t0, $t0, $s1   # $s0 holds the amount
       sc $t0, 0($s0)
       beq $t0, $zero, Retry
```

Synchronization: Additional Notes

- Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions.
- When building a synchronization library (if you have to), you must consider the memory consistency model and put proper memory barriers.
- Synchronization can be a *performance bottleneck.*