

CS/ECE 5381/7381
Computer Architecture
Spring 2023

Dr. Manikas
Computer Science
Lecture 19: Apr. 11, 2023

Thread-Level Parallelism

(Chapter 5, Hennessy and Patterson)

Note: some course slides adopted
from publisher-provided material

Outline

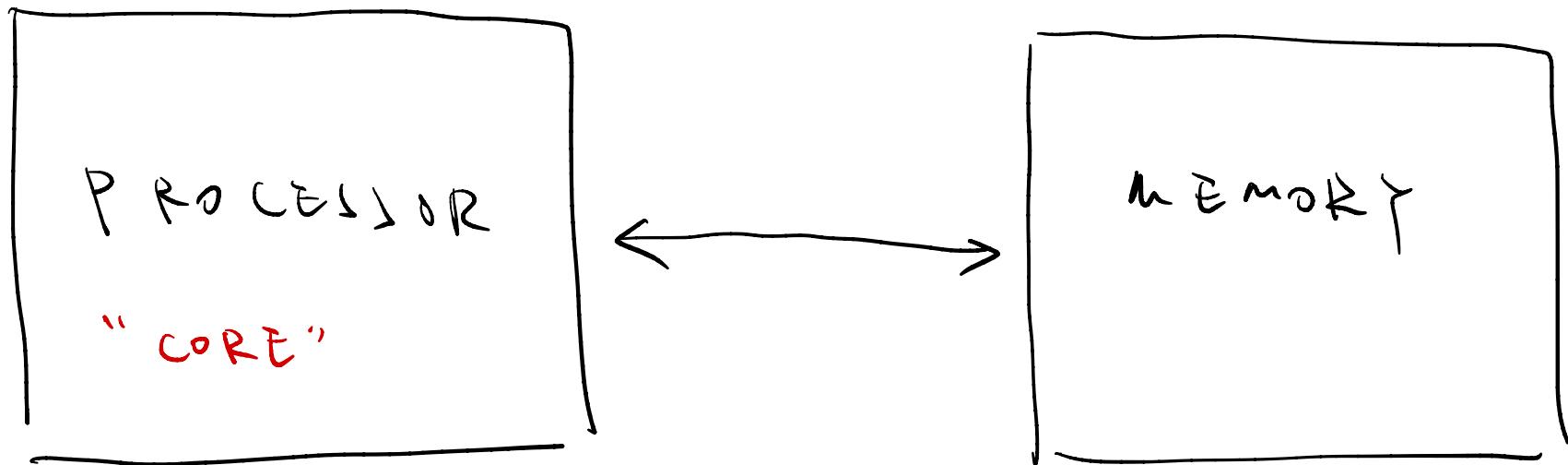
- 5.1 Introduction
- 5.2 Centralized Shared-Memory Architectures

Switch from Uniprocessors to Multiprocessors

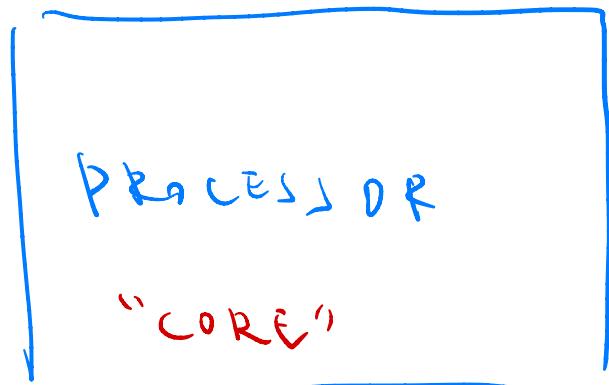
- Uniprocessor – MPU with one processor
- Multiprocessor – MPU with multiple processors (often called **cores**)
- Example: a “quadcore” MPU contains four cores
- To make efficient use of cores, program load must be split up such that parts of program can be run at the same time (**in parallel**)
- This is known as **parallel processing**, or **parallelism**.

RECALL FROM EARLIER:

ONE PROCESSOR: "UNIPROCESSOR"



"DUAL CORE" PROCESSOR



(CORE OF
ORIGINAL PROCESSOR)

TWO PROCESSORS:
"MULTIPROCESSOR".

Introduction

- Recall: **Data-Level Parallelism** used **SIMD**:
 - Single Instruction stream, Multiple Data streams
 - Single set of instructions accesses multiple sets of data
 - E.g., Vector Processors (Ch. 4)
- **Thread-Level Parallelism** uses **MIMD**:
 - Multiple Instruction streams, Multiple Data streams
 - Multiple sets of instructions access multiple sets of data

Terminology

- Thread = process to execute
 - Given n processors, program is often divided into n processes or **threads**
- Grain Size = amount of computation assigned to a thread

Comparison

- ILP: Instruction Level Parallelism (Ch. 3)
 - Extract parallelism in a single program
 - Out-of-order execution => instructions are sent to execution units based on instruction dependencies rather than program order
- Thread-level Parallelism
 - Multiple programs execute concurrently on multiple processors

Terminology

means you write it, and it doesn't change unless you change it.

- Program: static set of statements to perform computational steps of an algorithm
- Thread: embeds the execution of these steps
- Threads run on *cores*
 - Note: cores are sometimes called *CPU's* (Central Processing Units)



Hardware Multithreading

- **Multithreading** - The ability of an OS to support multiple, concurrent paths of execution within a single process
- The unit of resource ownership is referred to as a **process** or **task**
线程, 迅速切换.
- The unit of dispatching is referred to as a **thread** or **lightweight process**

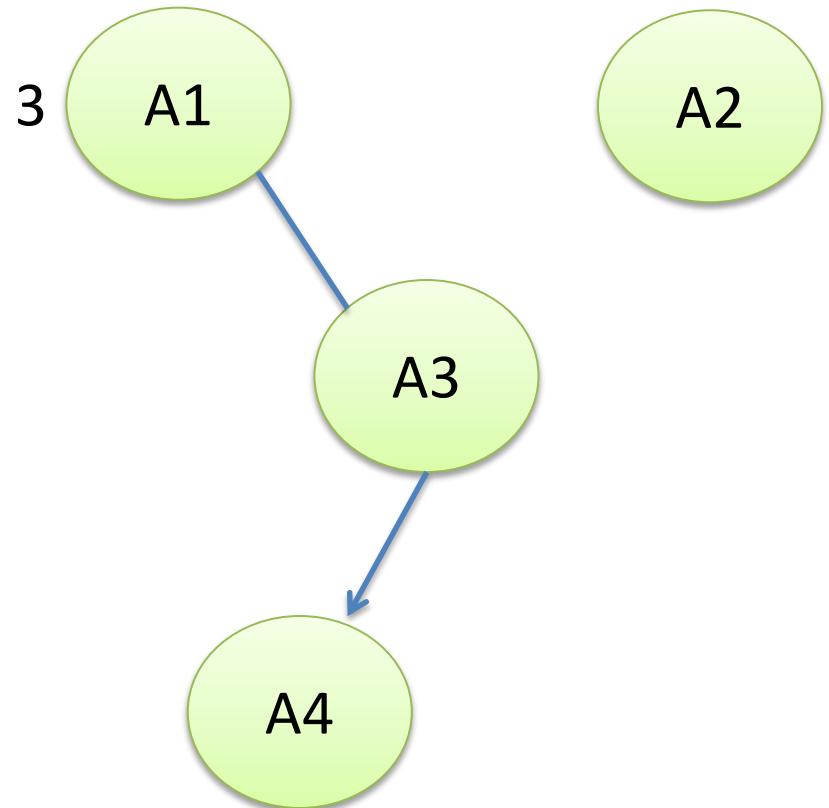
Multithreaded Processor Example

- Assume we have 2 threads X and Y with the following operations:

Thread X	Thread Y
A1 – takes 3 cycles to execute	B1 – takes 2 cycles to execute
A2 – no dependencies	B2 – conflicts for functional unit with B1
A3 – conflicts for functional unit with A1	B3 – depends on result of B2
A4 – depends on result of A3	B4 – takes 2 cycles to execute; no dependencies

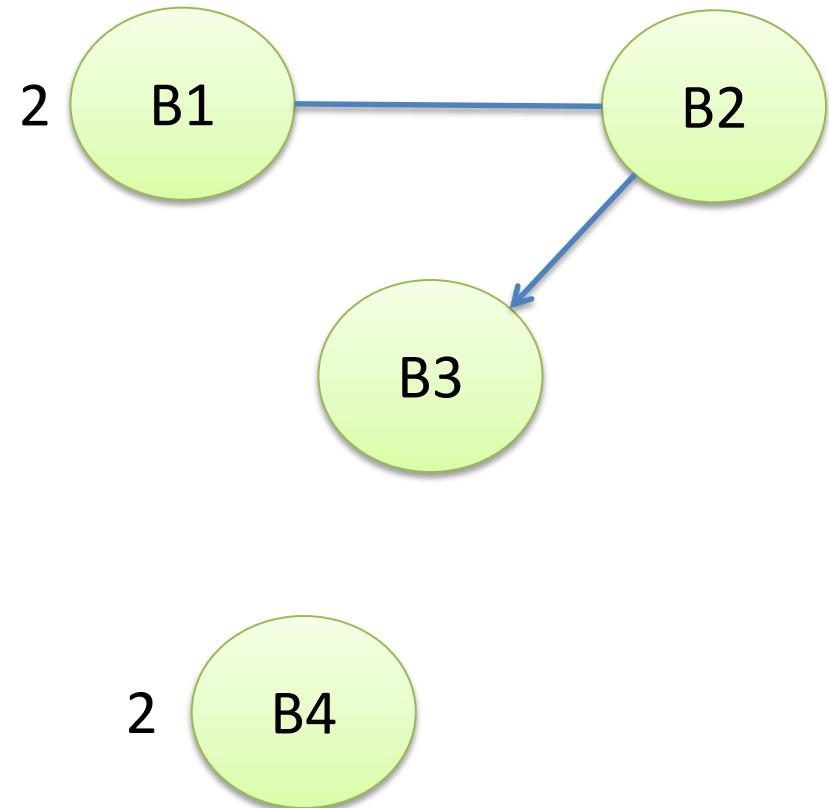
Dependency Graph for Thread X

Thread X
A1 – takes 3 cycles to execute
A2 – no dependencies
A3 – conflicts for functional unit with A1
A4 – depends on result of A3



Dependency Graph for Thread Y

Thread Y
B1 – takes 2 cycles to execute
B2 – conflicts for functional unit with B1
B3 – depends on result of B2
B4 – takes 2 cycles to execute; no dependencies

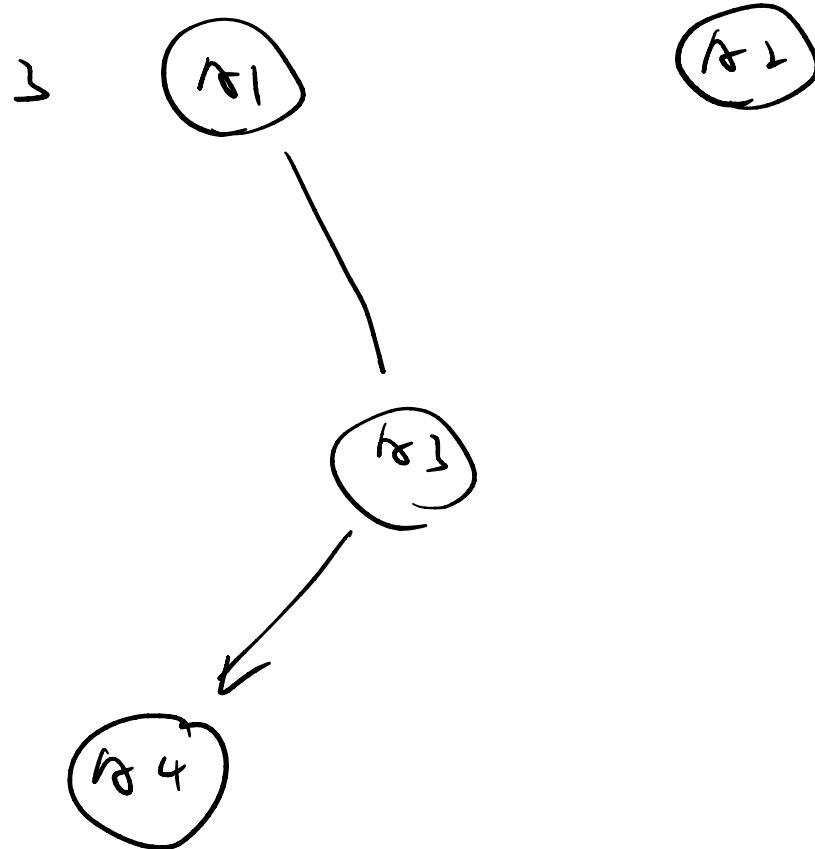


Example 5.1-1

- We have a dual-core processor that allows **out-of-order execution**:
 - Operations do not have to be executed in order (e.g. A1, A2, A3, A4), and can be done in *parallel*, as long as dependencies are observed
- Each core has a separate functional unit
- Only one thread can be run on a core at a time
- *What is the most efficient way (min. # of cycles) to execute threads X and Y on this processor?*

THE END

X



BASELINE

DO OPERATION
IN ORDER

<u>CYCLE</u>	<u>OPERATION</u>
1	A1
2	A1
3	A1
4	A2
5	A3
6	A4

NOTE FROM GRAPH

- α_2 CAN BE DONE IN PARALLEL
WITH any OTHER OPERATION
- α_1 AND α_3 CANNOT BE DONE IN PARALLEL
- α_3 AND α_4 CANNOT BE DONE IN PARALLEL
 - α_3 MUST BE DONE BEFORE α_4 .

REVISE TABLE

CYCLE	OPERATION
1	α_1, α_2
2	α_1
3	α_1
4	α_3
5	α_4

RECALL :

WE CAN EXECUTE

OPERATIONS OUT

OF ORDER

DO WE DO ANY OTHER

OPERATIONS IN PARALLEL?

α_1 AND α_4

TRY THIS

THREAD X

CYCLE OPERATION

1 α_3, α_2

2 α_4, α_1

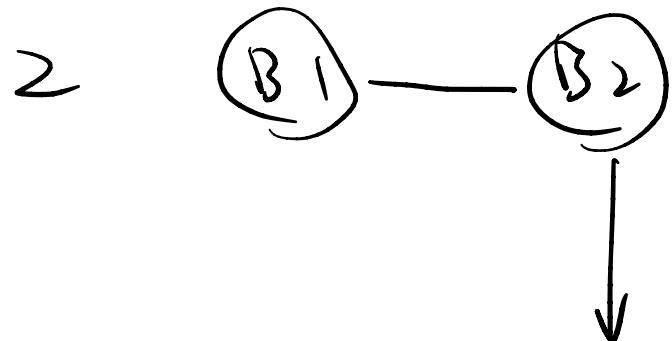
3 α_1

4 α_1

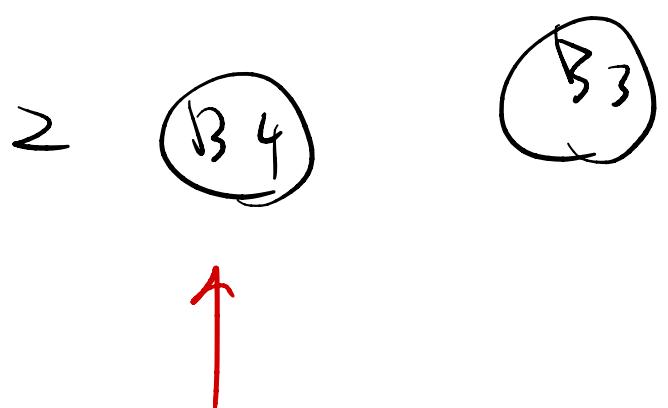
WENT FROM 6 CYCLES TO 4 CYCLES.

THREAD

- B₁, B₂ CANNOT DONE IN
PARALLEL



- B₂, B₃ CANNOT DONE IN
PARALLEL



- B₂ MUST BE DONE BEFORE
B₃

B₄ CAN BE DONE IN PARALLEL

WITH ANY OTHER OPERATION

B₁, B₃ can BE DONE IN PARALLEL

BASE LINE

<u>CYCLE</u>	<u>OPERATION</u>
1	B1
2	B1
3	B2
4	B3
5	B4
6	B4

REVISION

<u>CYCLE</u>	<u>OPERATION</u>
1	B1, B4
2	B1, B4
3	B2
4	B3

↓

<u>CYCLE</u>	<u>OPERATION</u>
1	B2, B4
2	B1, B3, B4
3	B1

Parallel Processing Challenges

- Limited parallelism available in programs
 - Affects amount of program execution speedup
- Relatively high cost of communications between processor units
 - Latencies – long communication delays

Parallel-Programming Model

- Defines how *parallel* computations can be expressed in a *high-level* programming language
 - Simple extensions through an API (application-programming interface) of common programming languages such as C/C++
- Threads or processes must communicate and coordinate their activity (synchronization)

同步

Code segment S1

```
for (i=0; i<N; i++) {  
    sum = sum + A[i];  
    B[i] = C[i] + 2;  
}
```

Code segment S2

```
for (k=0; k<N; k++) {  
    B[k] = B[k] * 4;  
}
```

- Currently, we have sequential code running on one processor
- Code segment S1 executes before S2

Code segment S1

```
for (i=0; i<N; i++) {  
    sum = sum + A[i];  
    B[i] = C[i] + 2;  
}
```

Code segment S2

```
for (k=0; k<N; k++) {  
    B[k] = B[k] * 4;  
}
```

- What if we want to run S1 and S2 in *parallel*?
 - Dual-core processor – S1 on core 1, S2 on core 2
- S1 must avoid writing into variables that are input to S2
 - RAW hazard for vector B
- S2 must avoid writing into variables that are input (WAR) or output (WAW) for S1 – also affects vector B

DATA vs FUNCTION PARALLELISM

- DATA PARALLELISM:
 - PARTITION THE DATA SET
 - APPLY THE SAME FUNCTION TO EACH PARTITION
 - SPMD (SINGLE PROGRAM MULTIPLE DATA)
 - (NOT SIMD)
 - MASSIVE PARALLELISM
- FUNCTION PARALLELISM
 - INDEPENDENT FUNCTIONS ARE EXECUTED ON DIFFERENT PROCESSORS
 - E.G.: STREAMING--SOFTWARE PIPELINE
 - MORE COMPLEX, LIMITED PARALLELISM
- COMBINE BOTH: FUNCTION + DATA PARALLELISM

EXAMPLE: MATRIX MULTIPLY + SUM

SEQUENTIAL PROGRAM:

```
1     sum = 0;  
2     for (i=0, i<N, i++)  
3         for (j=0, j<N, j++) {  
4             C[i,j] = 0;  
5             for (k=0, k<N, k++)  
6                 C[i,j] = C[i,j] + A[i,k]*B[k,j];  
7             sum += C[i,j];  
8         }
```

- MULTIPLY MATRICES $A[N,N]$ BY $B[N,N]$ AND STORE RESULT IN $C[N,N]$
- ADD ALL ELEMENTS OF C

$$N \begin{bmatrix} \textcolor{red}{x} \textcolor{blue}{x} \\ C \end{bmatrix} = \begin{bmatrix} \textcolor{blue}{A} \end{bmatrix} \times \begin{bmatrix} \textcolor{red}{B} \end{bmatrix}$$

MATRIX MULTIPLY

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} \begin{bmatrix} b_{12} \\ b_{22} \end{bmatrix}$$

$$C_{11} = a_{11}b_{11} + a_{12}b_{21} \quad \text{Can do IN PARALLEL}$$

$$C_{12} = a_{11}b_{12} + a_{12}b_{22}$$

$$C_{21} = a_{21}b_{11} + a_{22}b_{21}$$

$$C_{22} = a_{21}b_{12} + a_{22}b_{22}$$

Execution Time for Parallel Programs

- Execution time E is a function of:
 - Number of cores N
 - Thread start time (for each core) T
 - Program execution time (for a single core system)
 P

$$E = NT + \frac{P}{N}$$

Example 5.1-2

- Assume we have the matrix multiplication program shown in the previous slide
 - For a *single-core system*, the matrix multiplication operation takes 10 seconds
- We want to run this program on a *dual-core system*
 - Thread start time for each core is 200 ms
- What is the execution time for this program on the dual-core system?

PROGRAM EXECUTION TIME:

$$E = NT + \frac{P}{N} = 2(0.2) + \frac{10}{2} = 0.4 + 5 = 5.4 \text{ s}$$

N = NUMBER OF CORES = 2

T = THREAD START TIME = 200 ms

P = PROGRAM EXECUTION TIME = 10 s

(SINGLE CORE)

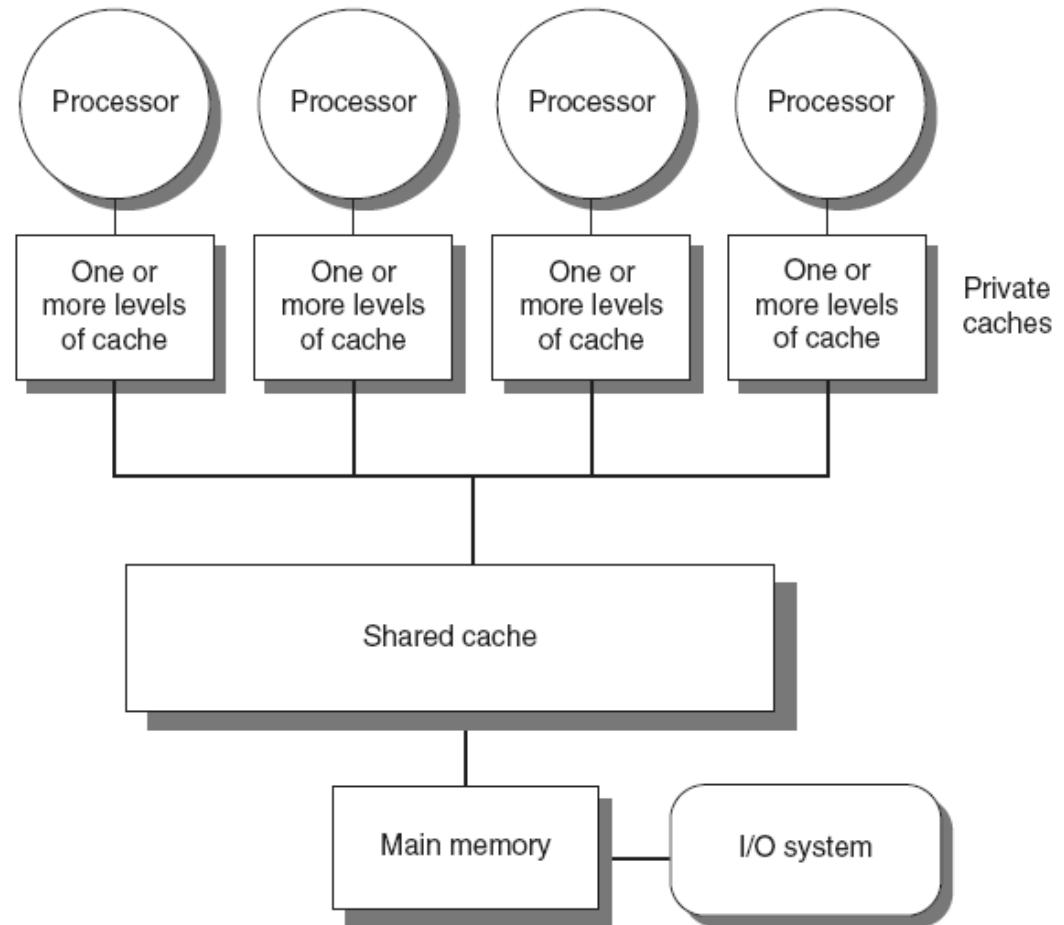
$T = 200 \text{ ms} = 0.2 \text{ s}$

Outline

- 5.1 Introduction
- 5.2 Centralized Shared-Memory Architectures

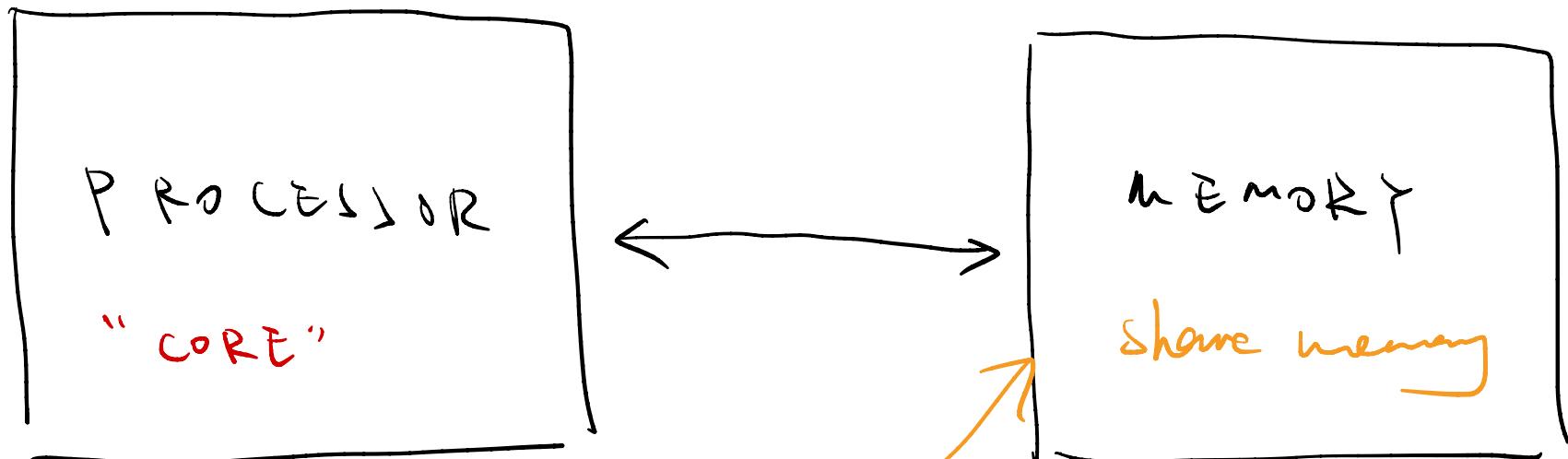
Shared-Memory Multiprocessors (SMP)

- Small number of cores
- Share single memory with uniform memory latency
- For large number of cores, use DSM

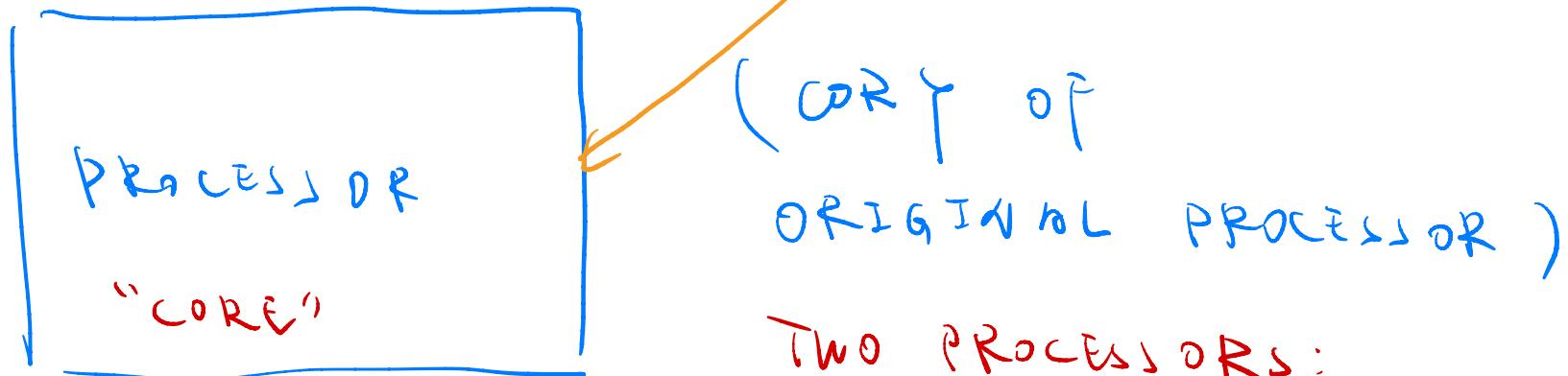


RECALL FROM EARLIER:

ONE PROCESSOR: "UNIPROCESSOR"



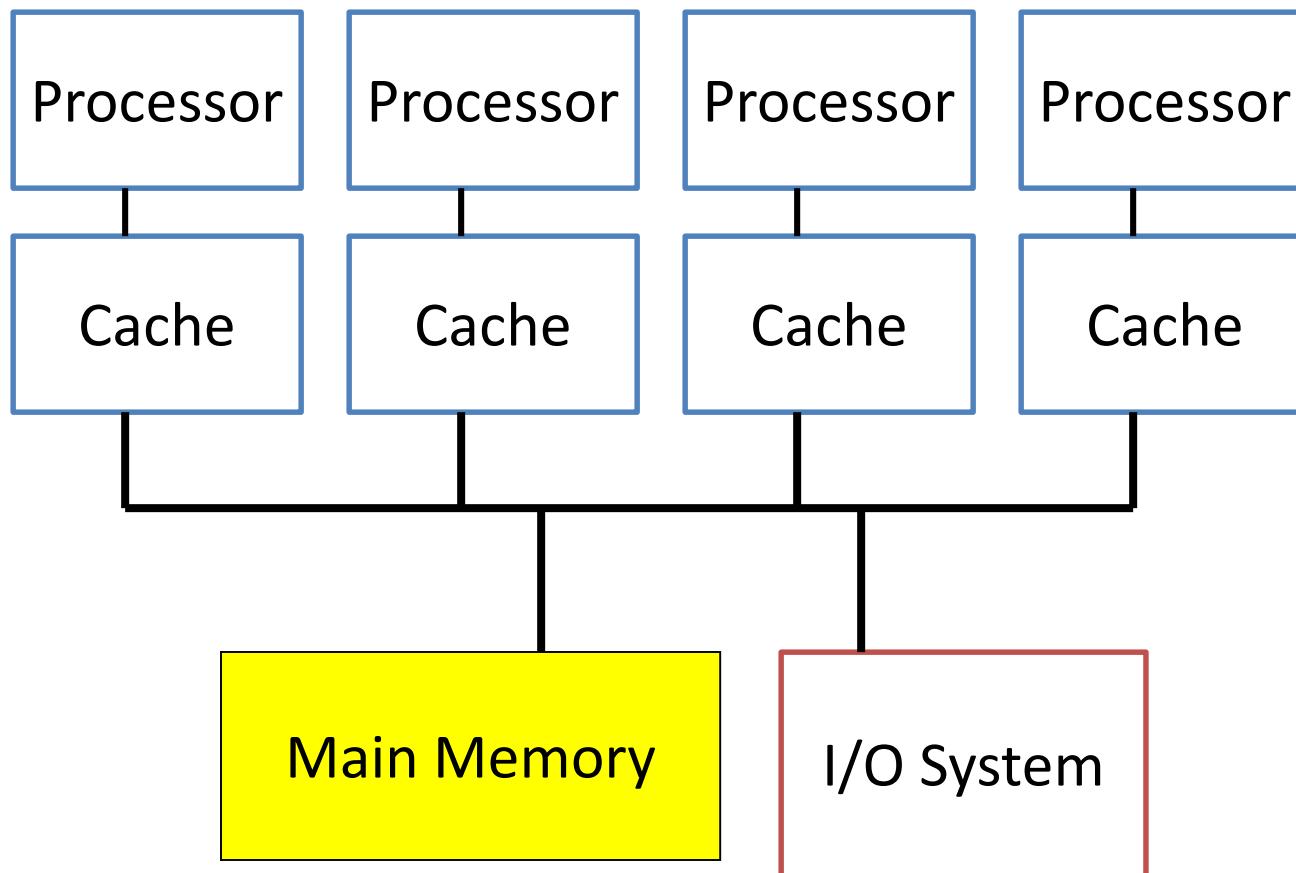
"DUAL CORE" PROCESSOR

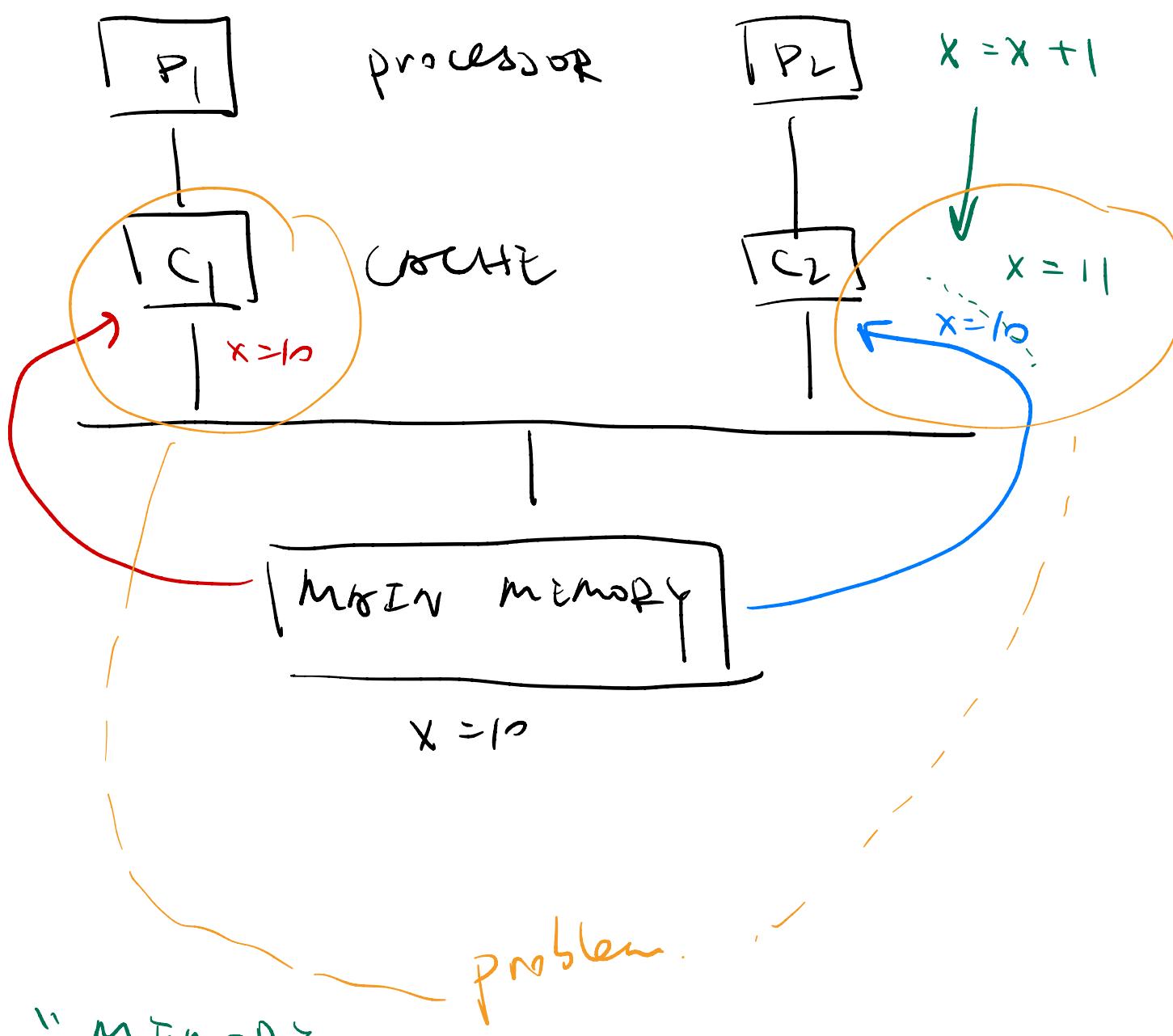


Centralized Shared-Memory Architectures

- Type of SMP
 - Small number of cores
 - Share single memory with uniform memory latency
 - Private caches only (recall earlier SMP types)

Centralized Shared-Memory





"Memory Consistency"
problem.

How to agree on value of x ?

Multiprocessor Cache Coherence

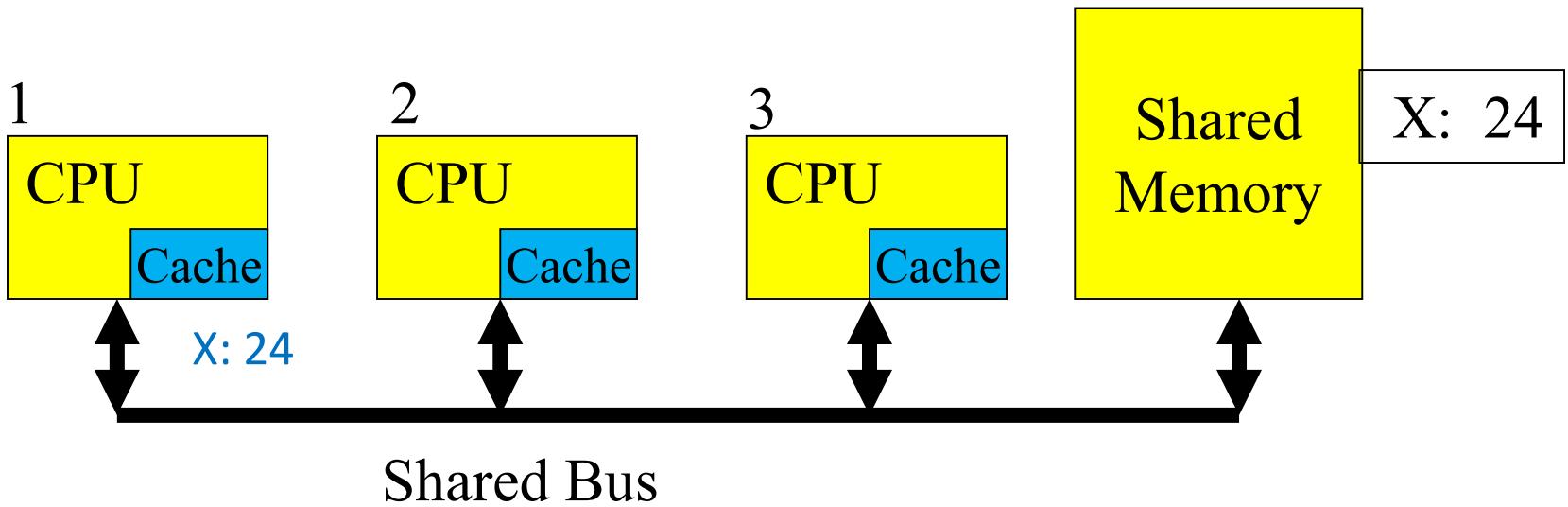
下节课 - 3.

- Recall that a processor's view of main memory is through the cache
- With multiple processors, each processor has its own cache
- How to make sure that the values in each cache match?
 - This is the cache coherence problem

Problem of Memory Coherence

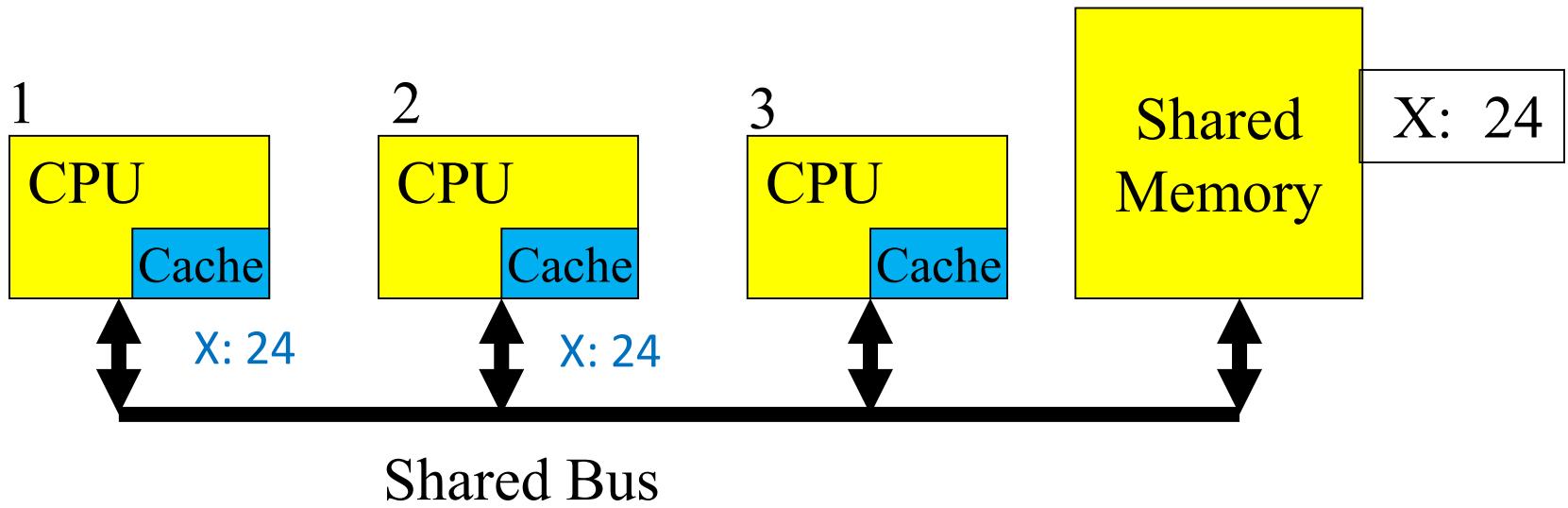
- Assume just single level caches and main memory
- Processor writes to location in its cache
- Other caches may hold shared copies - these will be out of date
- Updating main memory alone is not enough

Example



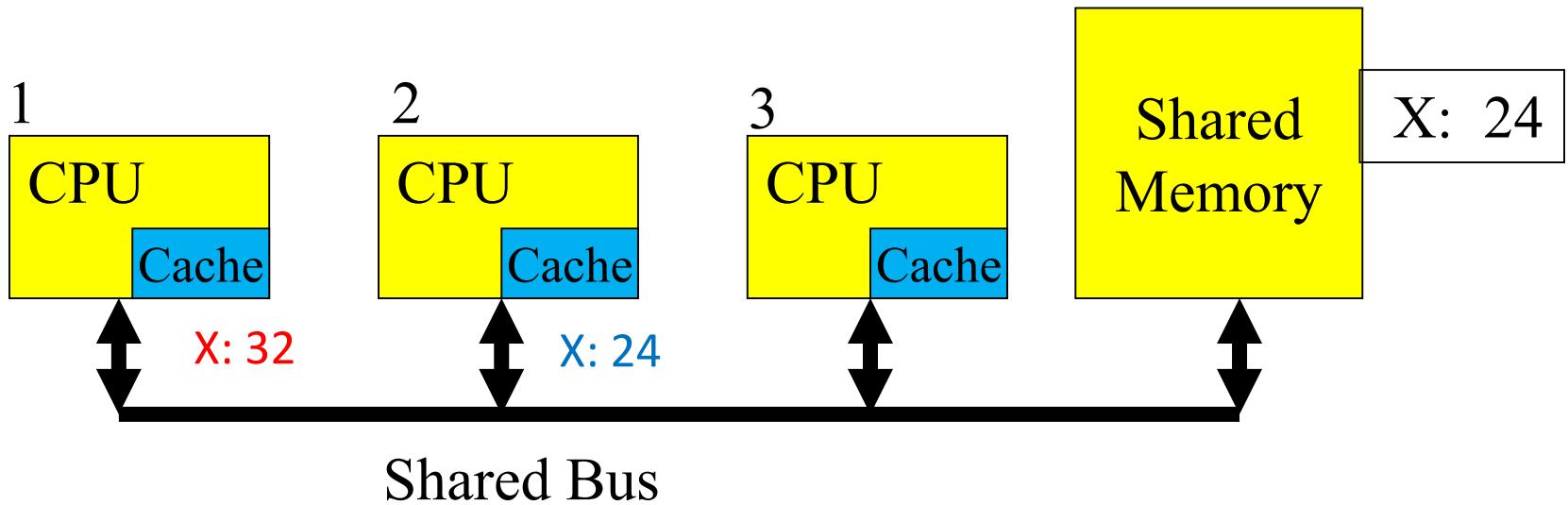
Processor 1 reads X: obtains 24 from memory and caches it

Example



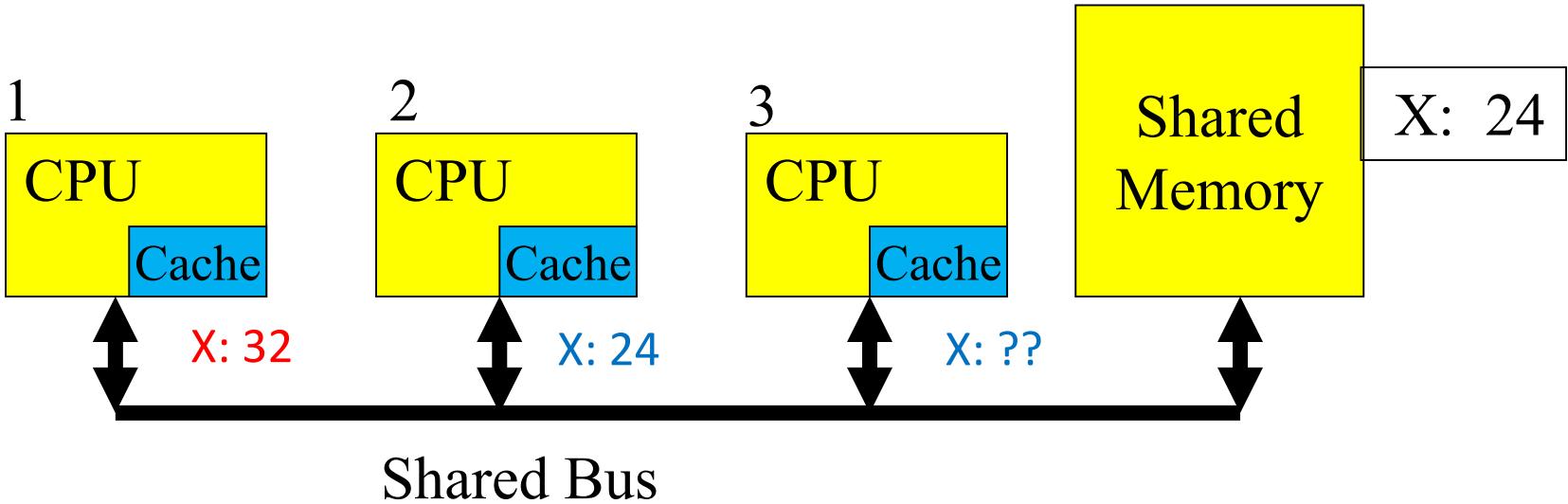
Processor 2 reads X: obtains 24 from memory and caches it

Example



Processor 1 writes 32 to X: its locally cached copy is updated

Example



Processor 3 reads X: what value should it get?

Memory and processor 2 think it is 24

Processor 1 thinks it is 32

Cache Coherence Protocols

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

Snoopy Protocol – NOT!



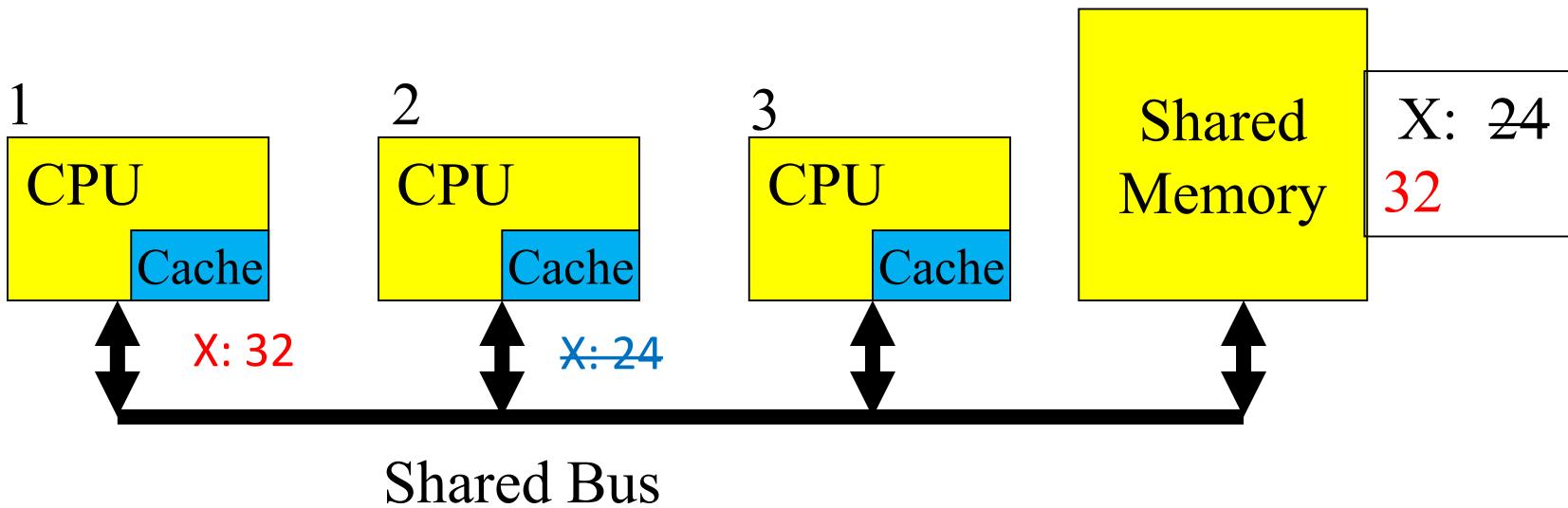
PEANUTS © United Feature Syndicate, Inc.

Snooping-Based Cache Coherence

- BASIC IDEA
 - TRANSACTIONS ON THE BUS ARE “VISIBLE” TO ALL PROCESSORS
 - BUS INTERFACE CAN “SNOOP” (MONITOR) THE BUS TRAFFIC AND TAKE ACTION WHEN REQUIRED
 - TO TAKE ACTION THE “SNOOPER” MUST CHECK THE STATUS OF THE CACHE

Snooping Protocol – Basic Scheme

- **Write Invalidate**
 - CPU wanting to write to an address, grabs a bus cycle and sends a ‘write invalidate’ message
 - All snooping caches invalidate their copy of appropriate cache line
 - CPU writes to its cached copy (assume for now that it also writes through to memory)
 - Any shared read in other CPUs will now miss in cache and re-fetch new data.



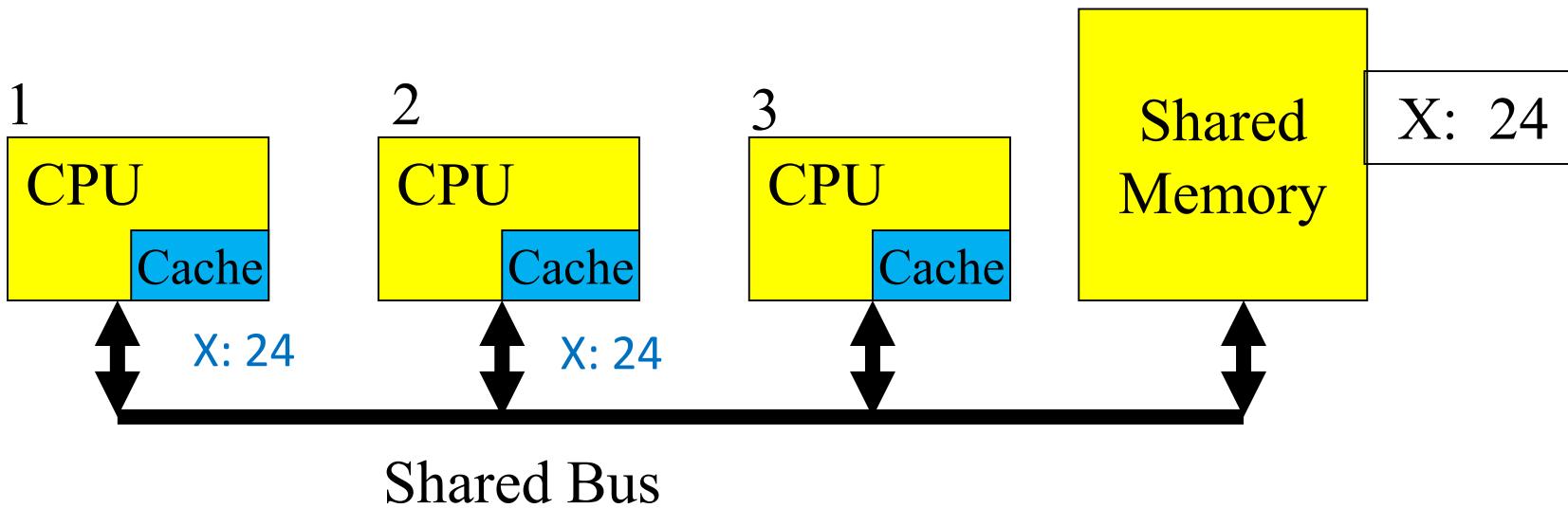
1. Processor 1 writes 32 to X: its locally cached copy is updated
2. All snooping caches *invalidate* their copy of X
3. CPU 1 writes cached copy to shared memory (new X value)
4. Any shared reads of X for other CPUs will now *miss* in local caches – must re-fetch new data from shared memory (X now is 32)

Implementation Issues

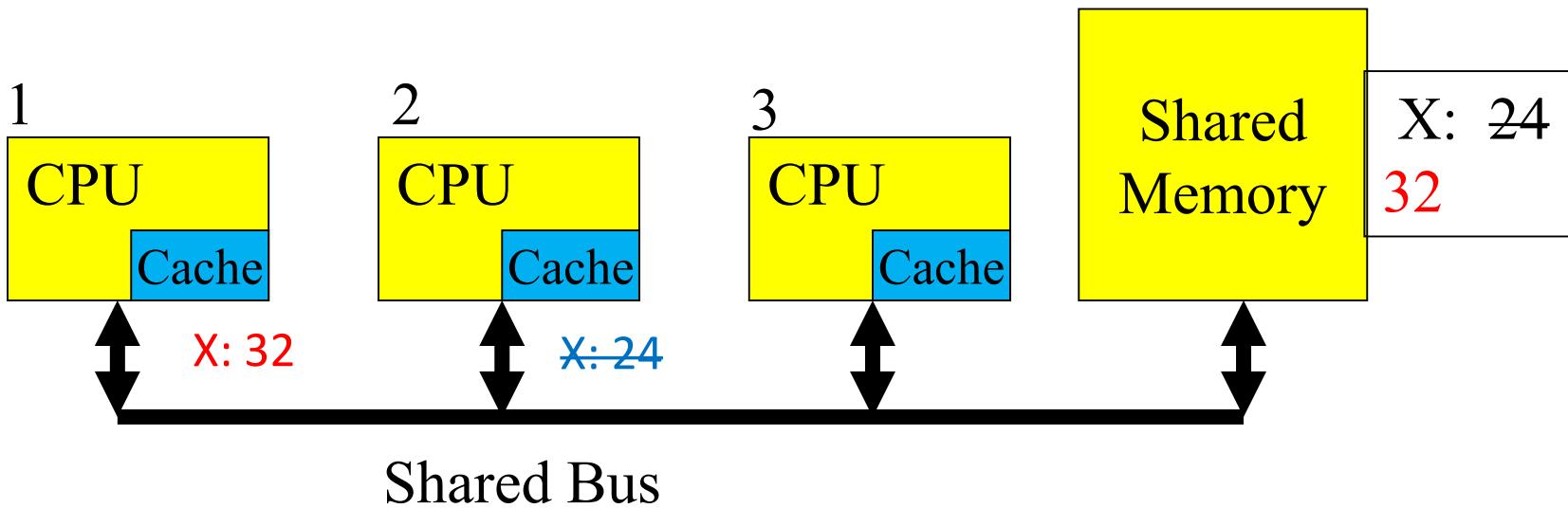
- Invalidate description assumed that a cache value update was written through to memory.
- If we used a ‘copy back’ scheme other processors could re-fetch old value on a cache miss.
- We need a protocol to handle all this.

MESI Protocol

- A practical multiprocessor invalidate protocol which attempts to minimize bus usage
- Extension of usual cache bits
 - Valid bit – data is valid in cache
 - Dirty bit – data is old, needs to be updated
- Allows usage of a ‘write back’ scheme - i.e. main memory not updated until ‘dirty’ cache line is displaced



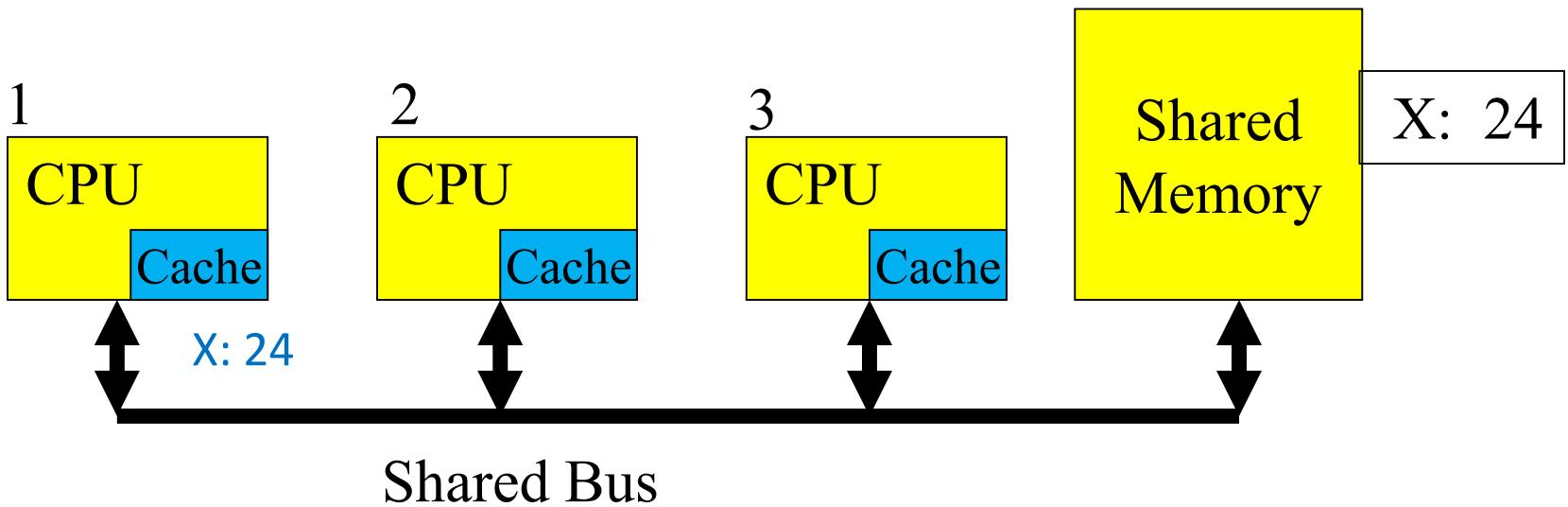
Cache	Valid	Dirty	Data
1	1	0	24
2	1	0	24
3	0	0	(empty)



Cache	Valid	Dirty	Data
1	1	0	32
2	1	1	24
3	0	0	(empty)

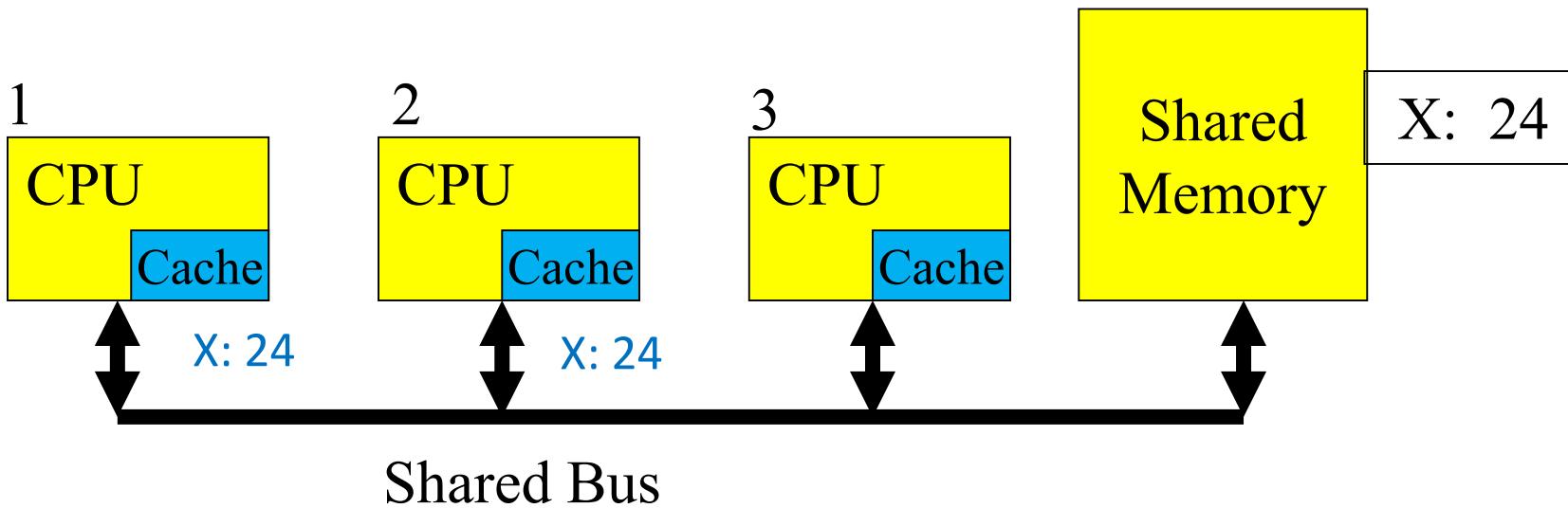
MESI Protocol – Cache Line States

- **(M) Modified** - cache line has been modified, is different from main memory - is the only cached copy. (multiprocessor ‘dirty’)
- **(E) Exclusive** - cache line is the same as main memory and is the only cached copy
stays in
- **(S) Shared** - Same as main memory but copies may exist in other caches.
- **(I) Invalid** - Line data is not valid (as in simple cache)



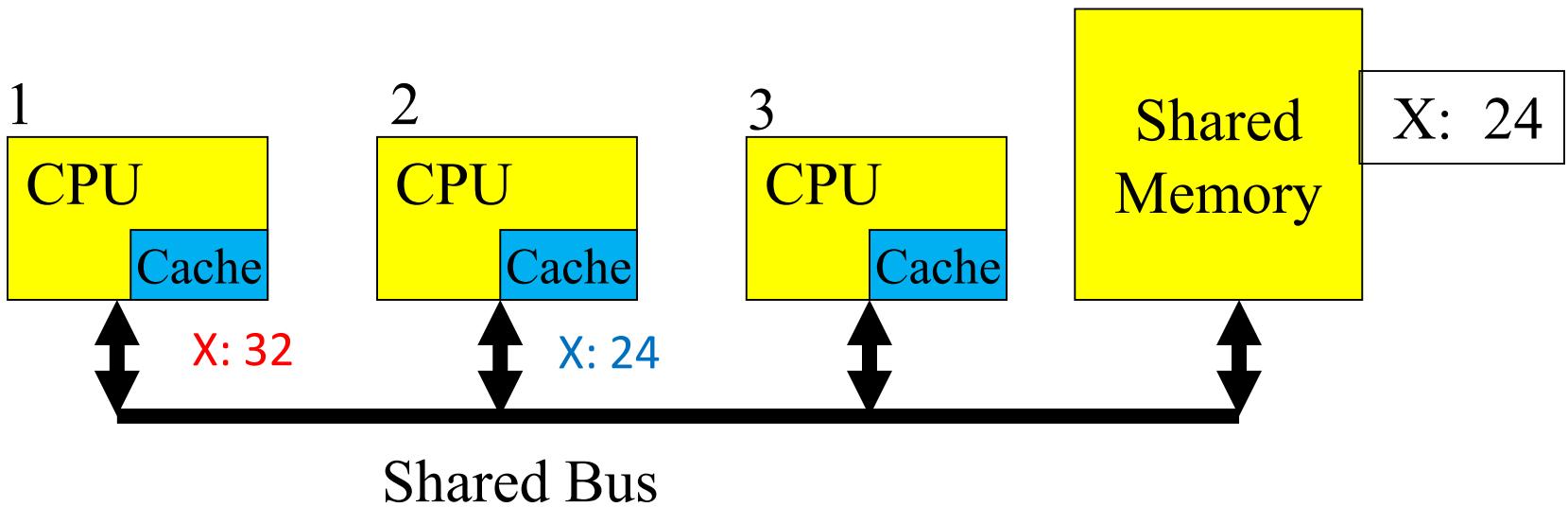
Cache 1 line is in **Exclusive (E)** state: same value as main (shared) memory for address X

Cache 2 and Cache 3 lines are in **Invalid (I)** state: no copies of data for address X



Cache 1 line is in **Shared (S)** state: same value as main memory for address X, but copies exist in other caches (Cache 2)

Note that Cache 2 line is also in Shared (S) state



Cache 1 line is in **Modified (M)** state: cache line value has been modified and is *different* from main memory