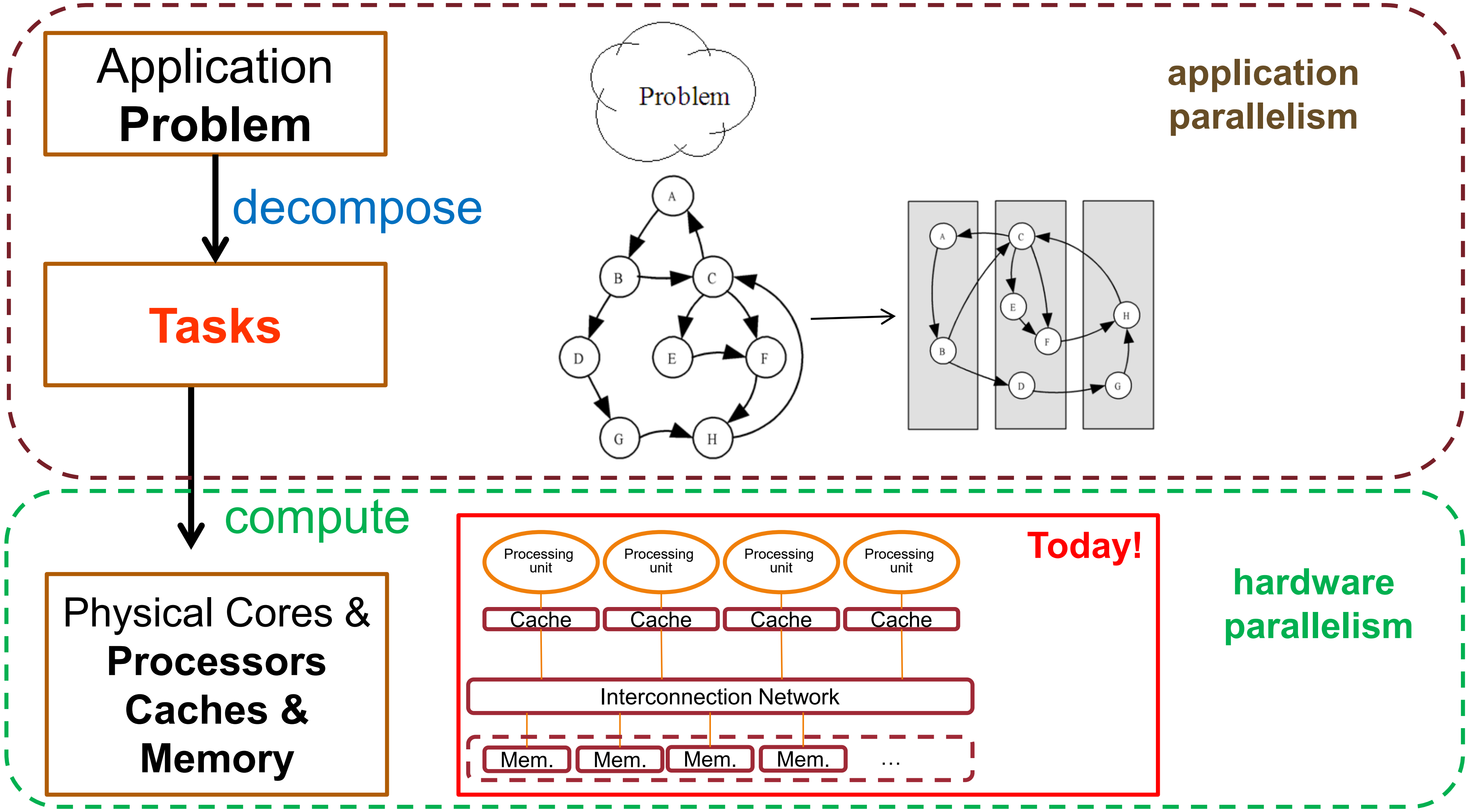
Cache Coherence
Memory Consistency

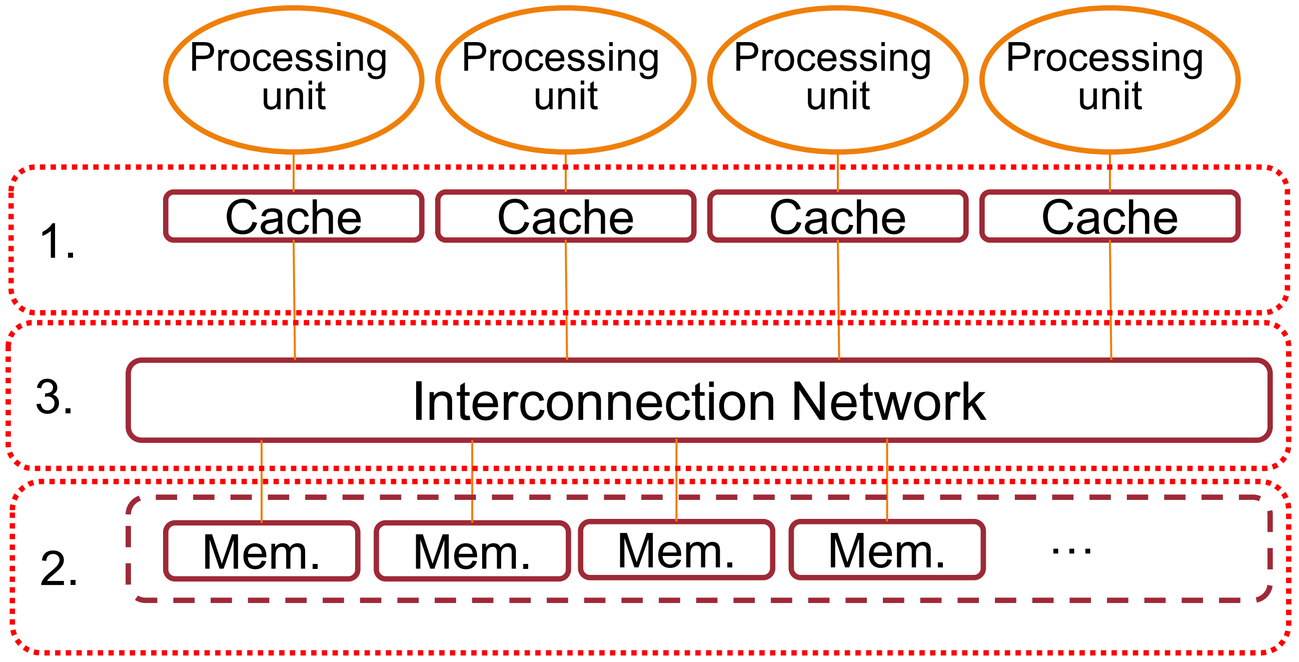
Lecture 07

Parallel Computing



Outline

1. Cache Coherence – L07
2. Memory Consistency – L07
3. Interconnection Networks
   - L11

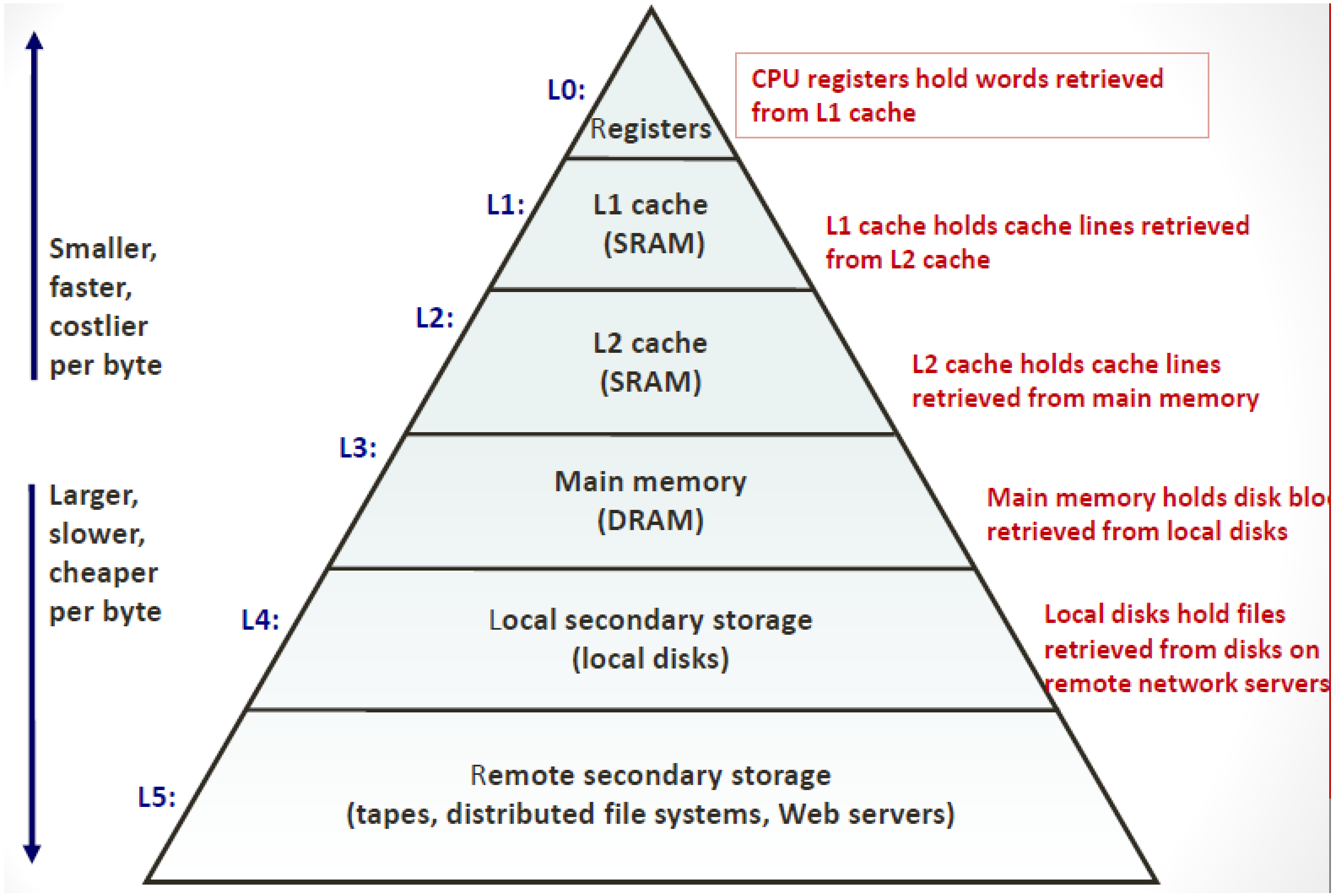


In shared address space, for correctly-synchronized programs!

Shared Address Space Model

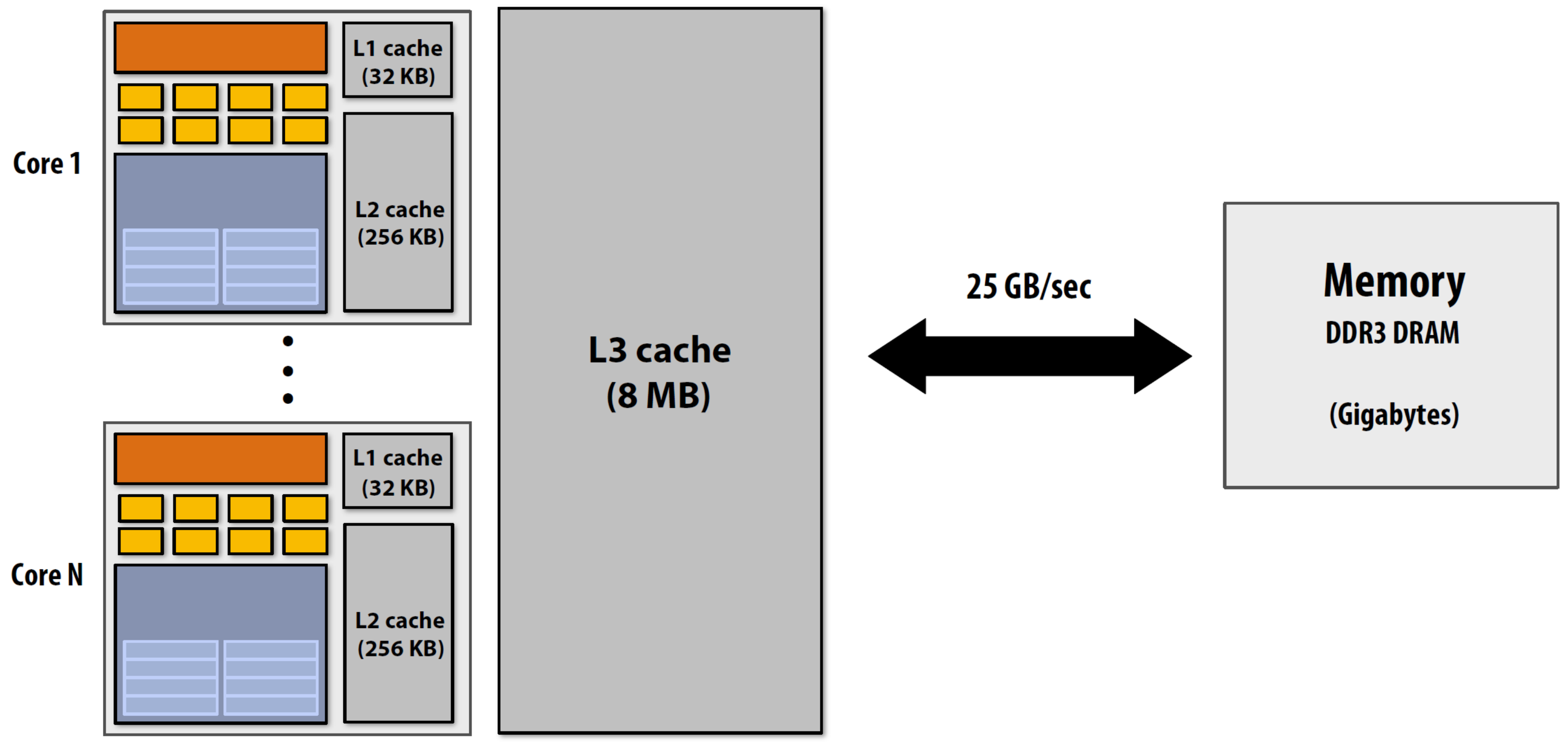
- Communication abstraction
  - Tasks communicate by reading/writing to shared variables
  - Ensure mutual exclusion via use of locks
  - Logical extension of uniprocessor programming
- Requires hardware support to implement efficiently
  - Any processing unit can load and store from any address
  - Even with NUMA, costly to scale

The Memory Hierarchy


Smaller, faster, costlier per byte
Larger, slower, cheaper per byte
L0:
Registers
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
Main memory (DRAM)
L4:
Local secondary storage (local disks)
L5:
Remote secondary storage (tapes, distributed file systems, Web servers)
CPU registers hold words retrieved from L1 cache
L1 cache holds cache lines retrieved from L2 cache
L2 cache holds cache lines retrieved from main memory
Main memory holds disk blo
retrieved from local disks
Local disks hold files retrieved from disks on remote network servers

Recap: why do modern processors have cache?

- Processors run efficiently when data is resident in caches
  - Caches reduce memory access latency
  - Caches provide high bandwidth data transfer to CPU



There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

CACHE COHERENCE

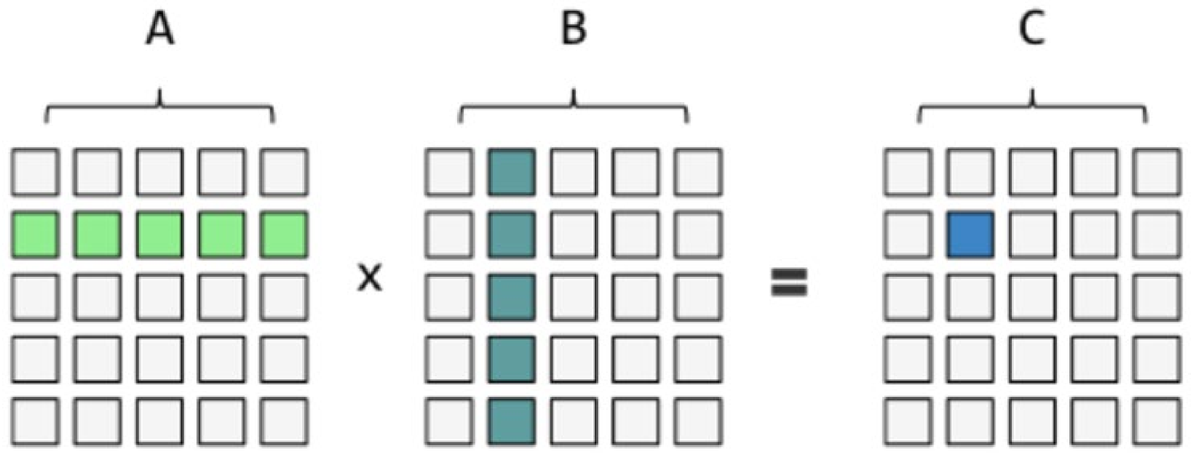
Cache Properties

- **Cache size:** larger cache increases access time (because of increased addressing complexity) but reduces cache misses

- **Block size (cache line):** data is transferred between main memory and cache in blocks of a fixed length
  - Larger blocks reduces the number of blocks but replacement lasts longer → block size should be small
  - But larger block increase the chance of spatial locality cache hit → block size should be large

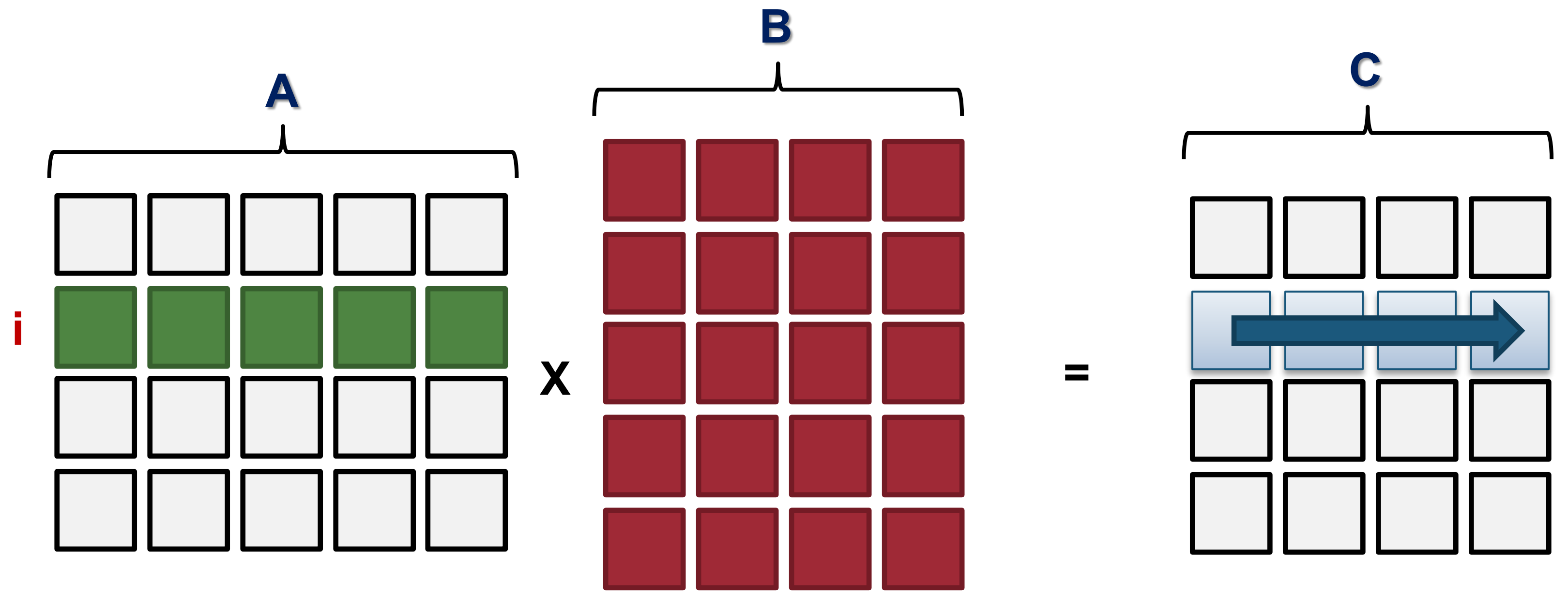
- Typical sizes for L1 cache blocks (lines) are 4 or 8 memory words (4 bytes)

Case Study: Matrix Multiplication



```
for i ← 0 to n-1
    for j ← 0 to n-1
        c[i, j] ← 0
        for k ← 0 to n-1
            c[i, j] ← c[i, j] + a[i, k] x b[k, j]
```

One Iteration



- Read:
  - Row ith of matrix **A**
  - Entire matrix **B**
- Write:
  - Row ith of matrix **C**

What are the potential cache related performance problems??

Matrix Multiplication and Cache

- Matrix Row-Column Ordering
  - Row-major vs Column-major

- Size of matrix:
  - Motivating question: How large is the square matrix that can be stored entirely in a 256KB cache?

  - You can assume double-precision floating point numbers (i.e. 8 bytes).

Write Policy

- Write-through - write access is immediately transferred to main memory
  - Advantage: always get the newest value of a memory block
  - Disadvantages: slow down due to many memory accesses
    - Use a write buffer
- Write-back - write operation is performed only in the cache
  - Write is performed to the main memory when the cache block is replaced
  - Uses a dirty bit
  - Advantages: less write operations
  - Disadvantages: memory may contain invalid entries

Write-back Cache

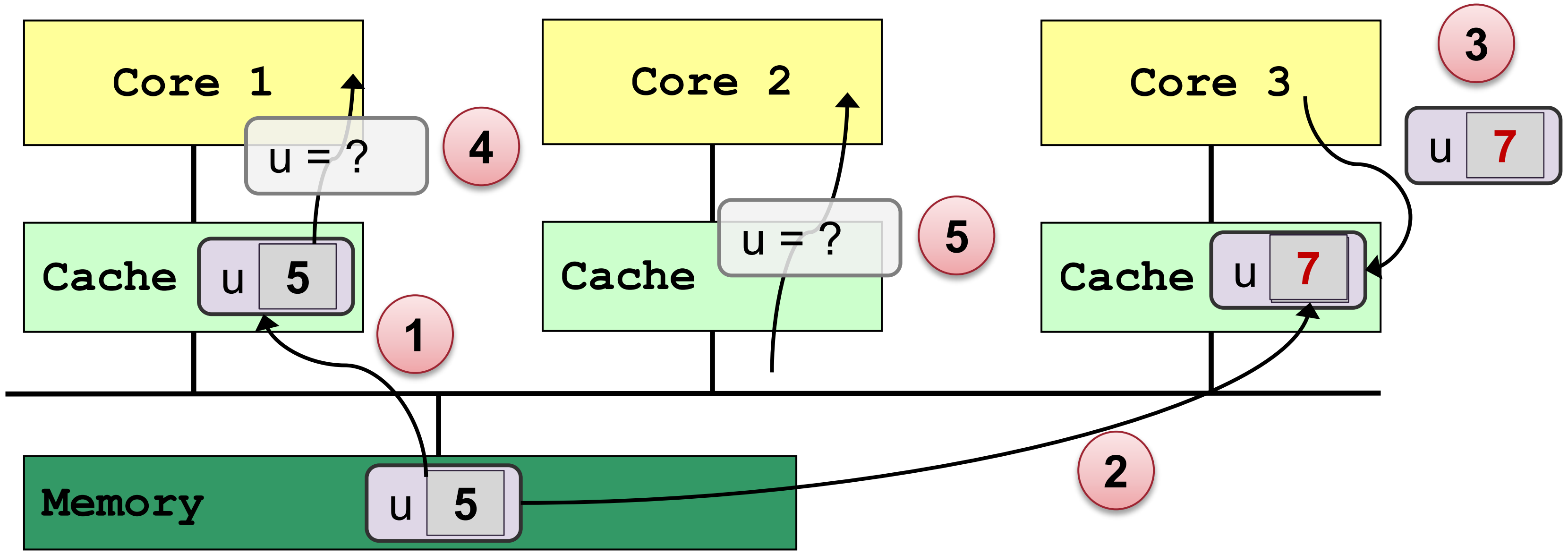
- Warm up: uniprocessor - processor with one core (no SMT)
- Example: processor executes `int x = 1;`

1. Processor performs write to address that "misses" in cache
2. Cache selects location to place line in cache, if there is a dirty line currently in this location, the dirty line is written out to memory
3. Cache loads line from memory ("allocates line in cache")
4. Whole cache line is fetched
5. Cache line is marked as dirty

Line state	Tag	Data (64 bytes)

Cache Coherence Problem

- Multiple copies of the same data exists on different caches
- Local update by processor → Other processors may still see the unchanged data



Memory Coherence for Shared Address Space

- Intuitively, reading value at an address should return the last value written at that address by any processing unit (core)
- Uniprocessor: typically writes come from one source, the processing unit
- Multiprocessor: a single shared address space leads to a memory coherence problem because there is
  - Global storage space (main memory)
  - Core caches (multiple levels)

Memory Coherence: Definition

- **Coherence** ensures that each processing unit has consistent view of memory (each memory location) through its local cache
  - All processing units must agree on the order of reads/writes to the **SAME memory location** (address), X
    - In other words: it is possible to put all operations involving the SAME memory location, X, on a timeline such that the observations of all processing units are consistent with that timeline
- Three properties of a *coherent* memory system
  - Write Propagation
  - Transaction Serialization

Memory Coherence: Properties

- Property One:
  - Given the sequence
    1. **`P (processing unit)`** writes to **`X`**
    2. No write to **`X`** (from the other processing units)
    3. **`P`** reads from **`X`**
  - **`P`** should get the value written in 1.
  - Known as **Program Order** property

Memory Coherence: Properties (2)

- Property Two:
  - Given a system that meets program order property, the sequence
    1. P_1 (processing unit) writes to `X`
    2. No further write to `X` (by any processing unit)
    3. P_2 reads from `X`:
  - P_2 should read value written by P_1

➔ Writes become visible to other processing units **eventually**

➢ Known as **Write Propagation** property

Memory Coherence: Properties (3)

- Property Three:
  - Given the sequence
    1. Write $\mathtt{v_1}$ to $\mathtt{x}$ (by any processing unit)
    2. Write $\mathtt{v_2}$ to $\mathtt{x}$ (by any processing unit)
  - processing units can **never** read $\mathtt{x}$ as $\mathtt{v_2}$, then **later** as $\mathtt{v_1}$

  ➔ All writes to a location (from the same or different processing units) are seen in the same order by all processing units

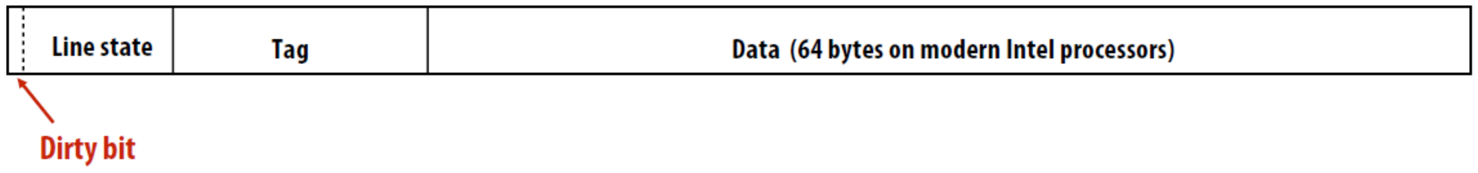
  ➢ Known as **Transaction Serialization** property

Maintaining Memory Coherence

- The programmer must ensure the programs are correctly synchronized
  - Programs suffer of data races when changes are made to the same memory location without proper synchronization
- Cache coherence can be maintained by:
  - Software based solution
    - OS + Compiler + Hardware aided solution
    - E.g. OS uses page-fault mechanism to propagate writes
  - Hardware based solution
    - Most common on multiprocessor system
    - Known as *cache coherence protocols*

Tracking Cache Line Sharing Status

- Two major categories:
  - **Snooping Based**
    - No centralized directory
    - Each cache keeps track of the sharing status
    - Cache *monitors* or *snoops* on the bus
      - to update the status of cache line
      - takes appropriate action
    - Most common protocol used in architectures with a bus
  - **Directory Based**
    - Sharing status is kept in a centralized location
    - Commonly used with NUMA architectures

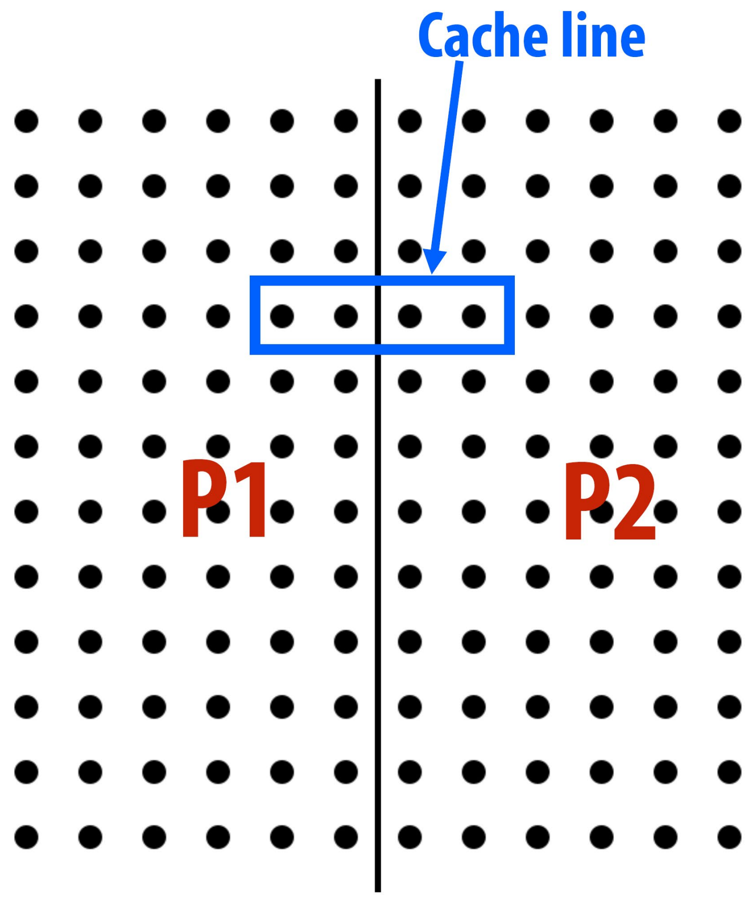


Snooping Cache Coherence

- Bus-based cache coherence
  - All the processing units on the bus can observe every bus transactions ➔ (Write Propagation)
  - Bus transactions are visible to the processing units in the same order ➔ (Write Serialization)
- The cache controllers:
  - "snoop" on the bus and monitor the transactions
  - Takes action for relevant bus transaction
    - i.e., if it has the memory block in cache
- Granularity of cache coherence is cache block

Cache Coherence Implications

- Overhead in shared address space
  - CC appears as increased memory latency in multiprocessor
  - CC lowers the hit rate in cache
- Cache ping-pong
  - Multiple processing units read and modify the same global variable
- False sharing
  - 2 processing units write to different addresses
  - The addresses map to the same cache line



Read, Write, Write, Read, Ok?

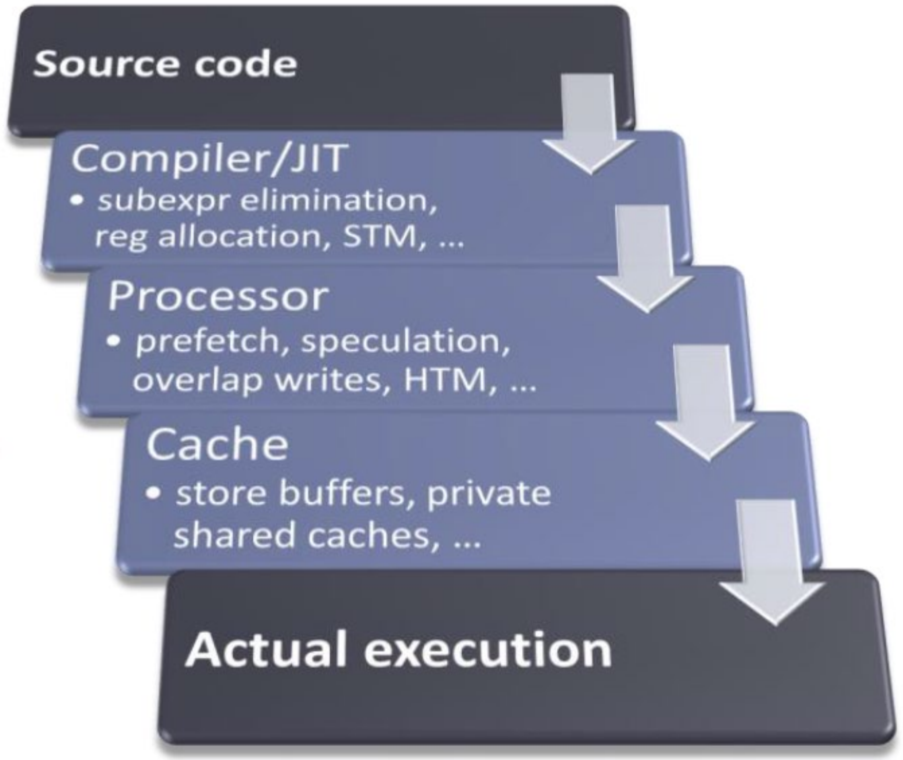
MEMORY CONSISTENCY MODELS

Coherence vs. Consistency

- **Coherence:** All processing units must agree on the order of reads/writes to the **SAME memory location** (address)
  - Coherence only guarantees that writes to address X will eventually propagate to other processing units
- Memory **consistency:** constraints the order in which memory operations performed by one thread become visible to other threads for **DIFFERENT memory locations**
  - Consistency deals with when writes to X propagate to other processing units, relative to reads and writes to other addresses

Importance of Memory Consistency

- Instructions of a program can be reordered to achieve better performance
  - The consistency models dictate what is allowed and what the expected behavior should be
  - Compilers and the hardware use consistency models at different levels to decide to produce equivalent programs
- The consistency model is used:
  - By programmers to reason about correctness and program behavior
  - By system/compiler designers to decide the reordering of memory operations possible by hardware and compiler


Source code
Compiler/JIT
• subexpr elimination, reg allocation, STM, …
Processor
• prefetch, speculation, overlap writes, HTM, …
Cache
• store buffers, private shared caches, …
Actual execution

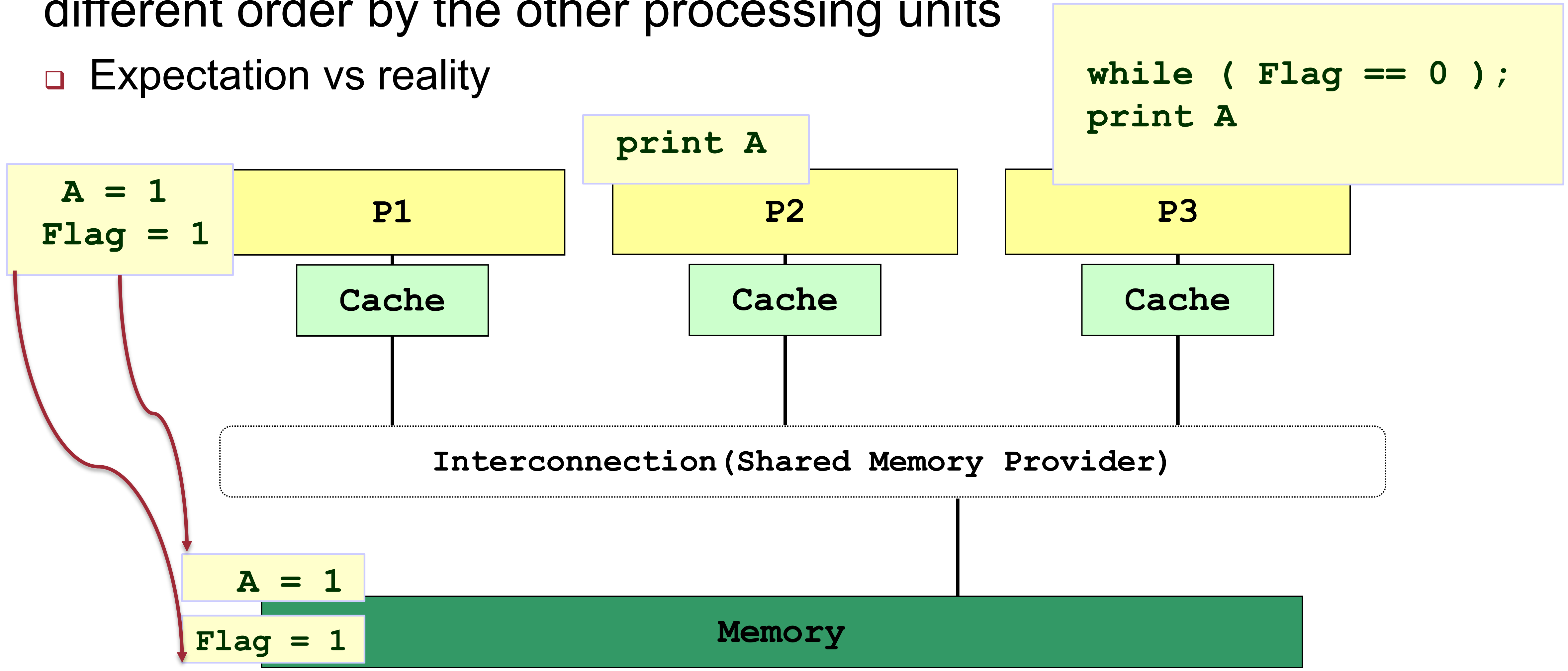
Memory Operations on Multiprocessors

- A program defines a sequence of loads and stores
  - "program order" of the loads and stores
- Four types of memory operation orderings
  - W→R: write to X must commit before subsequent read from Y *
  - R→R: read from X must commit before subsequent read from Y
  - R→W: read from X must commit before subsequent write to Y
  - W→W: write to X must commit before subsequent write to Y
- Reordered in an unintuitive way to hide write latencies
  - Application programmers rarely see this behavior
  - Systems (OS and compiler) developers see it all the time

*must commit ~ the results are visible (**observed**)

Memory Consistency Problem

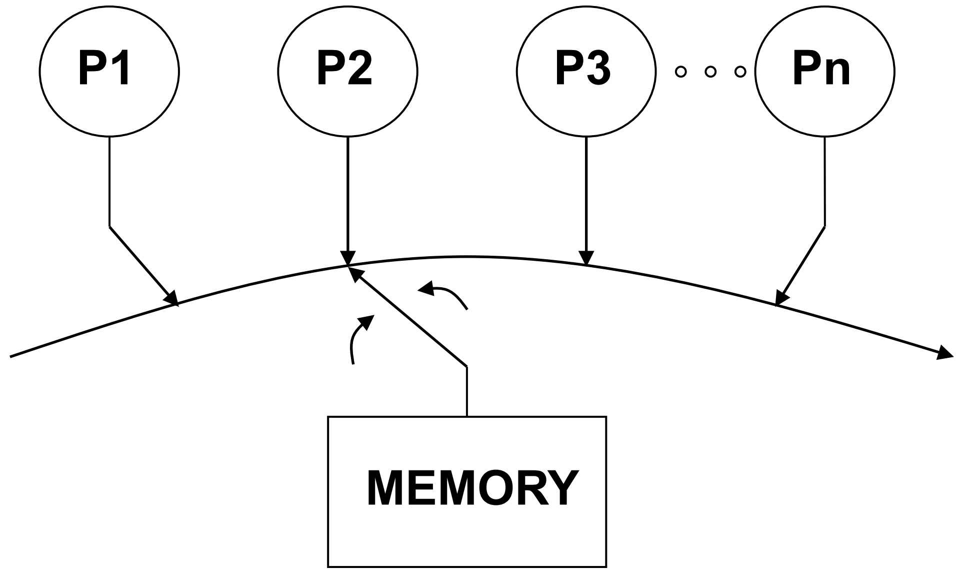
- The memory operations for A and Flag might be observed in different order by the other processing units
  - Expectation vs reality



Sequential Consistency Model (SC) – Lamport 1976

- A **sequentially consistent** memory system:
  - Every processing unit issues its memory operations in program order
    - **Global result of all memory accesses of all processing units appears to all processing units in the same sequential order** irrespective of the arbitrary interleaving of the memory accesses by different processing units
    - Effect of each memory operation must be visible to all processing units before the next memory operation on any processing unit
  - A sequentially consistent memory system preserves all four memory operation orderings (W→R, R→W, W→W, R→R)
- Extension of uniprocessor memory model
  - Intuitive, but can result in loss of performance

Sequential Consistency: Illustration



- **As if**:
  - One memory space
  - Only one memory operation at any point in time

Example 1: Sequential Consistent

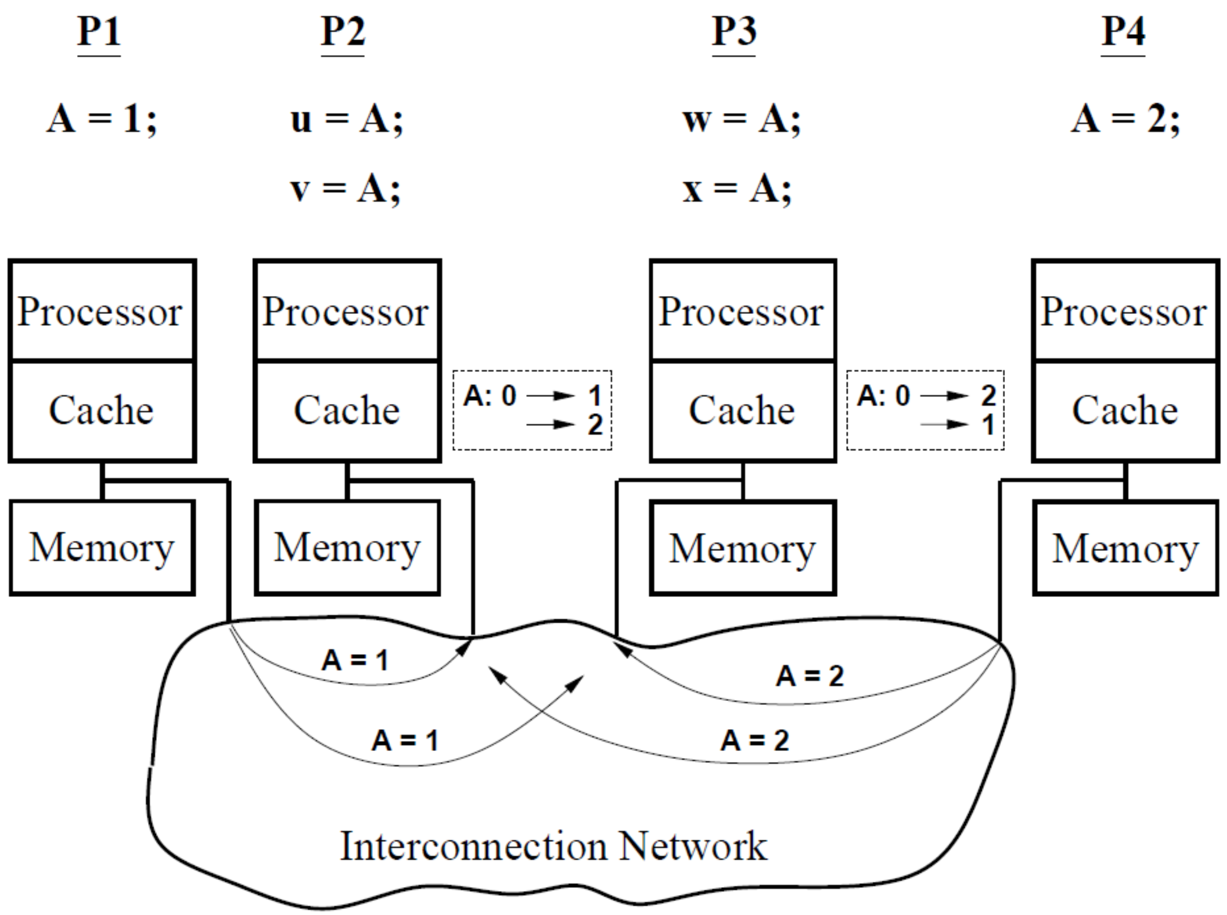
processor	P_1	P_2	P_3
program	(1) $x_1 = 1;$	(3) $x_2 = 1;$	(5) $x_3 = 1;$
	(2) print $x_2, x_3;$	(4) print $x_1, x_3;$	(6) print $x_1, x_2;$

- Initially, all variables are 0
- **Possible outputs**
  - (1)-(3)-(5)-**(2)-(4)-(6)**
    → **111111**
  - (1)-**(2)**-(3)-(4)-(5)-(6)
    → **001011**
- Memory accesses from different processors are arbitrarily interleaved!

- **Impossible outputs**
  - **011001**

Example 2: Sequential Consistency

- Initially A=0
- With sequential consistency
  - (u,v,w,x)=(1,2,2,1) should not be possible

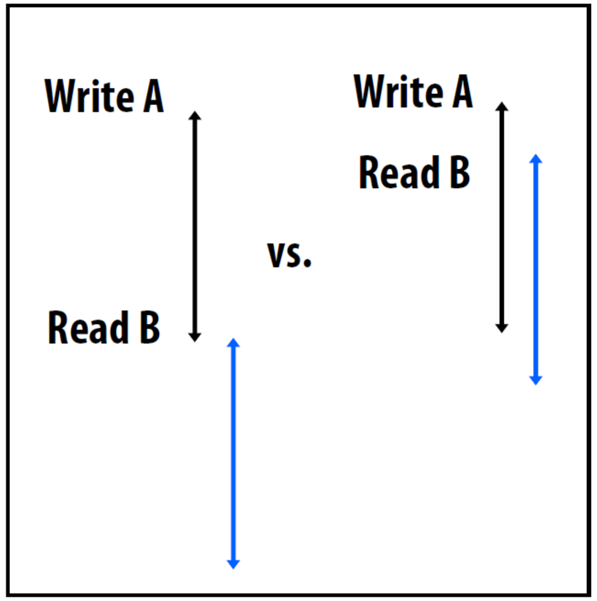


Relaxed Consistency

- Relaxed Consistency – relax the ordering of memory operations if data dependencies allow
  - Dependencies: if two operations access the **same memory location**
    - R → W: anti-dependency (WAR)
    - W → W: output dependency (WAW)
    - W → R: flow dependency (RAW)
    - Must be preserved!
- Relaxed memory consistency models allow certain orderings to be violated

Motivation for Relaxed Consistency

- Hide latencies!
  - To gain performance
  - Specifically, hiding memory latency: overlap memory access operations with other operations when they are independent
- How?
  - Overlap memory access operations with other operations when they are independent



Relaxed Memory Consistency Models

- Memory models resulted from relaxation of SC's requirements
- Program order relaxation:
  - Write → Read
  - Write → Write
  - Read → Read or Write
- All models provide overriding mechanism to allow a programmer to intervene

Relaxed Consistency: Write-to-read

- Key Idea:
  - Allow a read on processing unit **P** to be reordered w.r.t. to the previous write of the same processing unit
    - Hide the write latency
  - Different timing of the return of the read defines different models
  - **Data dependencies must be preserved!**
  - Cache coherence is preserved

Write-to-Read Program Order

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

- **T**otal **S**tore **O**rdering (**TSO**) :
  - Processing unit P can read B before its write to A is seen by all PUs

    (processing unit can move its own reads in front of its own writes)
  - Reads by other processing units cannot return new value of A until the write to A is **observed** by all processing units (**write atomicity** property)
- **P**rocessor **C**onsistency (**PC**) :
  - Return the value of any write (even from another processing unit) before the write is **observed** by all processing units
  - Write serialization and write propagation are preserved, but write atomicity is not
    - Writes are observed eventually by all processing units
    - Writes to the same memory location are observed by all PUs in the same order
    - But the writes can be read by some processing unit, before they are observed by all processing units

Example 1

P1

```
A = 1
Flag = 1
```

P2

```
while ( Flag == 0 );
print A
```

- A = Flag = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0 ✗
  - TSO/PC: A = 0 ✗

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Example 2

P1	P2

`A = 1`
`B = 1`
	`print B`
	`print A`

- A = B = 0 initially
- Can A = 0; B = 1 be printed under the models?
  - SC: A = 0; B = 1 ✗
  - TSO/PC: A = 0; B = 1 ✗

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Example 3

P1
```
A = 1
```

P2
```
while ( A == 0 );
B = 1
```

P3
```
while ( B == 0 );
print A
```

SC:
W -> R
R -> R
R -> W
W -> W

- A = B = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0 ✗
  - TSO: A = 0 ✗
  - PC: A = 0 √

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Example 4

P1

```
A = 1
print B
```

P2

```
B = 1
print A
```

SC:
W -> R
R -> R
R -> W
W -> W

- A = B = 0 initially
- Can A = 0; B = 0 be printed under the models?
  - SC: A = 0 ; B = 0 ✗
  - TSO: A = 0 ; B = 0 √
  - PC: A = 0 ; B = 0 √

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Write-to-Write Program Order

- Key Idea:
  - Writes can bypass earlier writes (to different locations) in write buffer
  - Allow write miss to overlap and hide latency

- Example Model:
  - **P**artial **S**tore **O**rdering (**PSO**)
    - Relax W → R order
      - Similar to TSO
    - Relax W → W order

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Example 5

P1

```
A = 1
Flag = 1
```

P2

```
while ( Flag == 0 );
print A
```

- A = Flag = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0 ✗
  - TSO/PC: A = 0 ✗
  - PSO: A = 0 √

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Example 6

P1

```
A = 1
B = 1
```

P2

```
print B
print A
```

- A = B = 0 initially
- Can A = 0; B = 1 be printed under the models?
  - SC: A = 0; B = 1 ✗
  - TSO/PC: A = 0; B = 1 ✗
  - PSO: A = 0; B = 1 √

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Example 7

P1

```
A = 1
```

P2

```
while ( A == 0 );
B = 1
```

P3

```
while ( B == 0 );
print A
```

- A = B = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0 ✗
  - TSO: A = 0 ✗
  - PC: A = 0 √
  - PSO: A = 0 ✗

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Example 8

P1

```
A = 1
print B
```

P2

```
B = 1
print A
```

- A = B = 0 initially
- Can A = 0; B = 0 be printed under the models?
  - SC: A = 0 ; B = 0 ✗
  - TSO: A = 0 ; B = 0 √
  - PC: A = 0 ; B = 0 √
  - PSO: A = 0 ; B = 0 √

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

For Your Exploration

- Weak ordering models
  - No completion order of the memory operations is guaranteed
  - i.e. relax R → R, R → W -> out-of-order execution
  - Provide additional synchronization operations
    - Lock/unlock
    - Memory fence (barrier)

Summary

- Cache coherence
  - Ensures that each processing unit has consistent view of memory through its local cache
- Memory consistency
  - Order of memory accesses → opportunity for reducing program execution time