Consistency Model

- A consistency model is contract between a distributed data store and processes, in which the processes agree to obey certain rules in contrast the store promises to work correctly.
- A consistency model basically refers to the degree of consistency that should be maintained for the shared memory data.
- If a system supports the stronger consistency model, then the weaker consistency model is automatically supported, but the converse is not true.
- The types of consistency models are **Data-Centric** and client centric consistency models.

Why Consistency Models Matter

- Each thread accesses two types of memory locations
  - Private: only read/written by that thread – should conform to sequential semantics
    - “Read A” should return the result of the last “Write A” in program order
  - Shared: accessed by more than one thread – what about these?
- Answer is determined by the **Memory Consistency Model** of the system
- Determines the order in which shared-memory accesses from different threads can “appear” to execute
  - In other words, determines what value(s) a read can return
  - More precisely, the set of all writes (from all threads) whose value can be returned by a read

Difference between Cache Coherence and Memory Consistency

Cache coherence	Memory consistency
Cache Coherence describes the behavior of reads and writes to the same memory location.	Memory consistency describes the behavior of reads and writes in relation to other locations.
Cache coherence required for cache-equipped systems.	Memory consistency required in systems that have or do not have caches.
Coherence is the guarantee that caches will never affect the observable functionality of a program	Consistency is the specification of correctness for memory accesses,
It is concerned with the ordering of writes to a single memory location.	It handles the ordering of reads and writes to all memory locations
A memory system is coherent if and only if – Can serialize all operations to that location – Read returns the value written to that location by the last store.	Consistency is a feature of a memory system if – It adheres to the rules of its Memory Model. – Memory operations are performed in a specific order.

Types of Consistency

- **Strict Consistency Model:** "The strict consistency model is the strongest form of memory coherence, having the most stringent consistency requirements. A shared-memory system is said to support the strict consistency model if the value returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations. That is, all writes instantaneously become visible to all processes."
- **Sequential Consistency Model:** "The sequential consistency model was proposed by **Lamport**. A shared-memory system is said to support the sequential consistency model if all processes see the same order of all memory access operations on the shared memory. The exact order in which the memory access operations are interleaved does not matter. If one process sees one of the orderings of three operations and another process sees a different one, the memory is not a sequentially consistent memory."
    - **Example:** Assume three operations read(R1), write(W1), read(R2) performed in an order on a memory address. Then (R1,W1,R2),(R1,R2,W1),(W1,R1,R2)(R2,W1,R1) are acceptable provided all processes see the same ordering.
- **Causal Consistency Model:** "The causal consistency model relaxes the requirement of the sequential model for better concurrency. Unlike the sequential consistency model, in the causal consistency model, all processes see only those memory reference operations in the same (correct) order that are potentially causally related. Memory reference operations that are not potentially causally related may be seen by different processes in different orders."
    - If a write(w2) operation is causally related to another write (w1) the acceptable order is (w1, w2).

Types of Consistency

- **FIFO Consistency Model:** For FIFO consistency, Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.
  - **Example**: If (w11) and (w12) are write operations performed by p1 in that order and (w21),(w22) by p2. A process p3 can see them as [(w11,w12),(w21,w22)] while p4 can view them as [(w21,w22),(w11,w12)].
- **Pipelined Random-Access Memory (PRAM) Consistency Model:** The pipelined random-access memory (PRAM) consistency model provides a weaker consistency semantics than the (first three) consistency models described so far. It only ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed as if all the write operations performed by a single process are in a pipeline. Write operations performed by different processes may be seen by different processes in different orders.
- **Weak Consistency Model:** Synchronization accesses (accesses required to perform synchronization operations) are sequentially consistent. Before a synchronization access can be performed, all previous regular data accesses must be completed. Before a regular data access can be performed, all previous synchronization accesses must be completed. This essentially leaves the problem of consistency up to the programmer. The memory will only be consistent immediately after a synchronization operation.

Types of Consistency

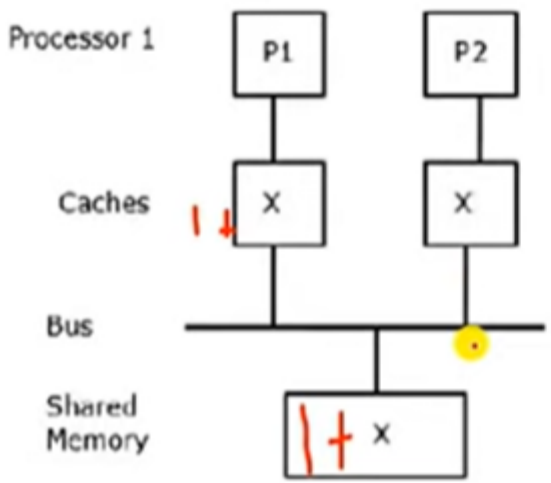
- **Release Consistency Model:** Release consistency is essentially the same as weak consistency, but synchronization accesses must only be processor consistent with respect to each other. Synchronization operations are broken down into acquire and release operations. All pending acquires (e.g., a lock operation) must be done before a release (e.g., an unlock operation) is done. Local dependencies within the same processor must still be respected. "Release consistency is a further relaxation of weak consistency without a significant loss of coherence.
- **Entry Consistency Model:** variants of release consistency, it requires the programmer (or compiler) to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared data item to be associated with some synchronization variable, such as a lock or barrier. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. When an acquire is done on a synchronization variable, only those data guarded by that synchronization variable are made consistent.
- **Processor Consistency Model:** Writes issued by a processor are observed in the same order in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. That is, two simultaneous reads of the same location from different processors may yield different results.
- **General Consistency Model:** A system supports general consistency if all the copies of a memory location eventually contain the same data when all the writes issued by every processor have completed.

What Does Coherence Mean?

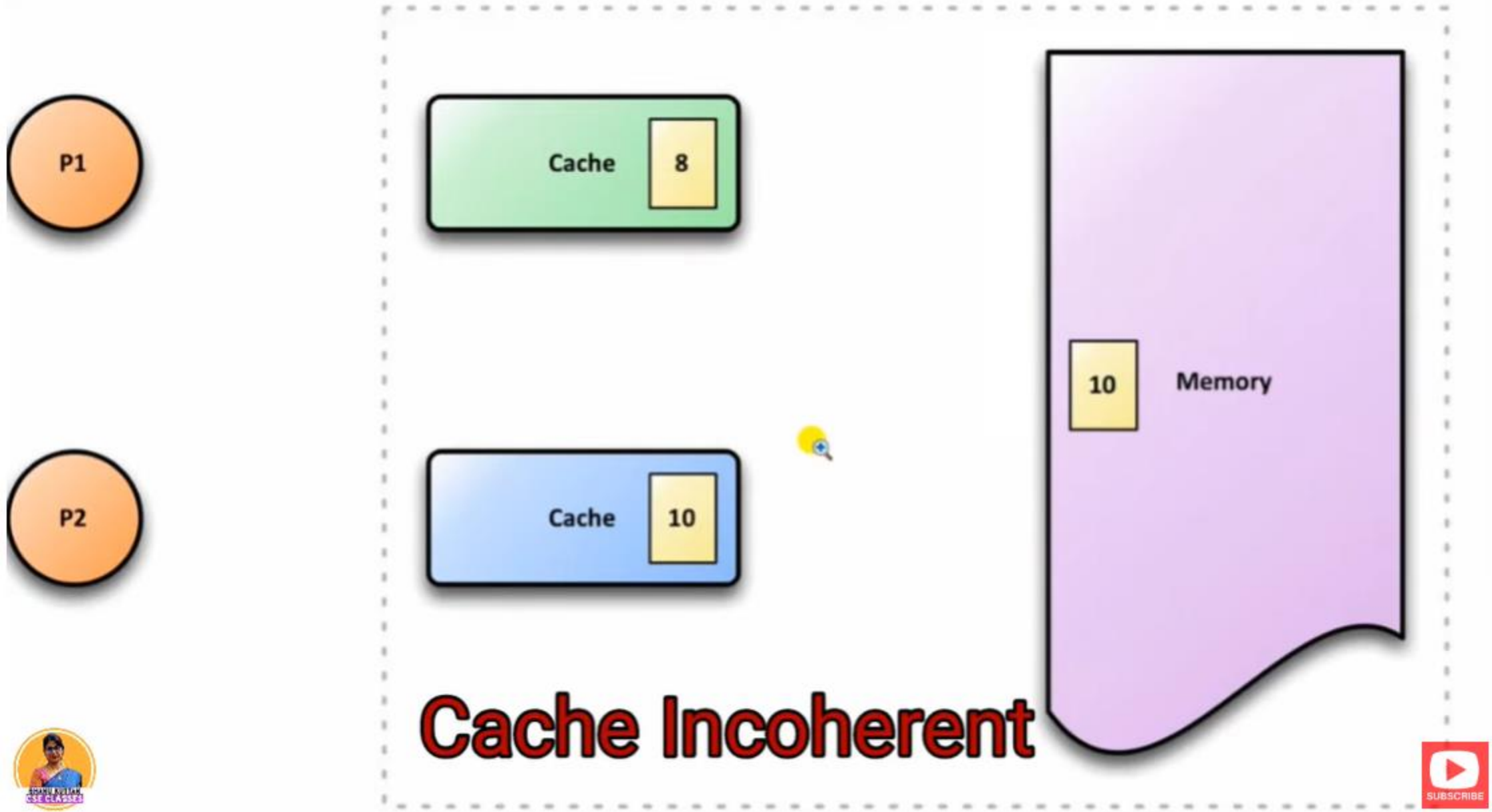
- Informally: – Any read must return the most recent write
  - Too strict and very difficult to implement
- Better: – Any write must eventually be seen by a read
  - All writes are seen in proper order (“serialization”)
- Two rules to ensure this:
  - If P writes x and P1 reads it, P’s write will be seen by P1 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)

Cache Coherence

CPU → C → M

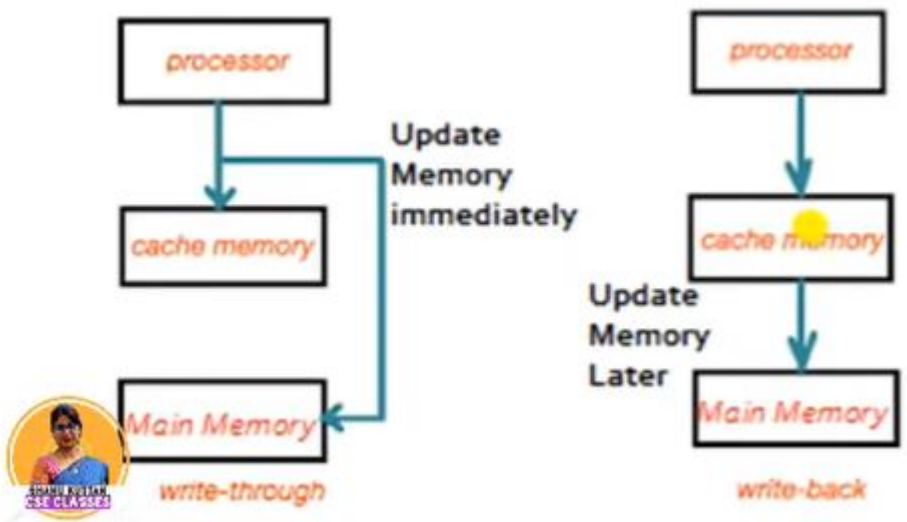


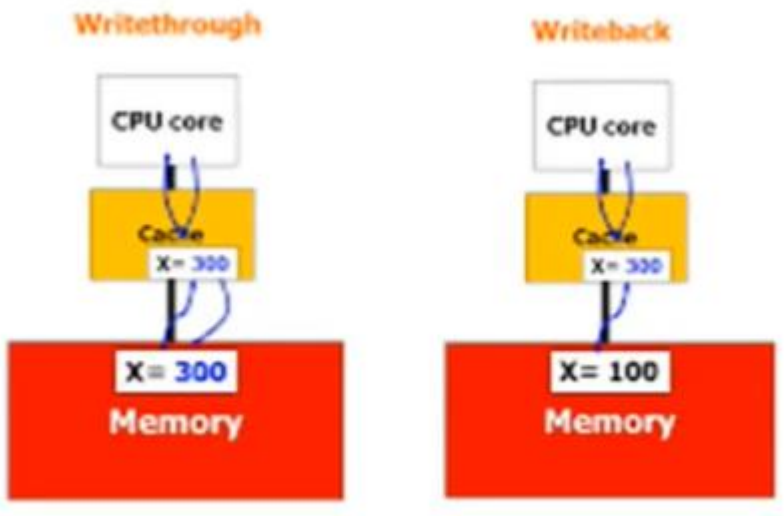
- For higher performance in a multiprocessor system,
  - each processor is usually have its own cache.
- In a multiprocessor system, ***data inconsistency*** may occur among ***adjacent levels or within the same level of the memory hierarchy.***
  - For example, the cache and the main memory may have inconsistent copies of the same object.
- As *multiple processors* operate in parallel, and independently multiple caches may possess different copies of the same memory block, this creates **cache coherence problem.**
- ***Cache coherence*** refers to the ***problem* of keeping the *data* in these *caches consistent*. The main problem is dealing with *writes* by a processor.**


P1
Cache
8
P2
Cache
10
10
Memory
Cache Incoherent
SUBSCRIBE

Cache Write Policies

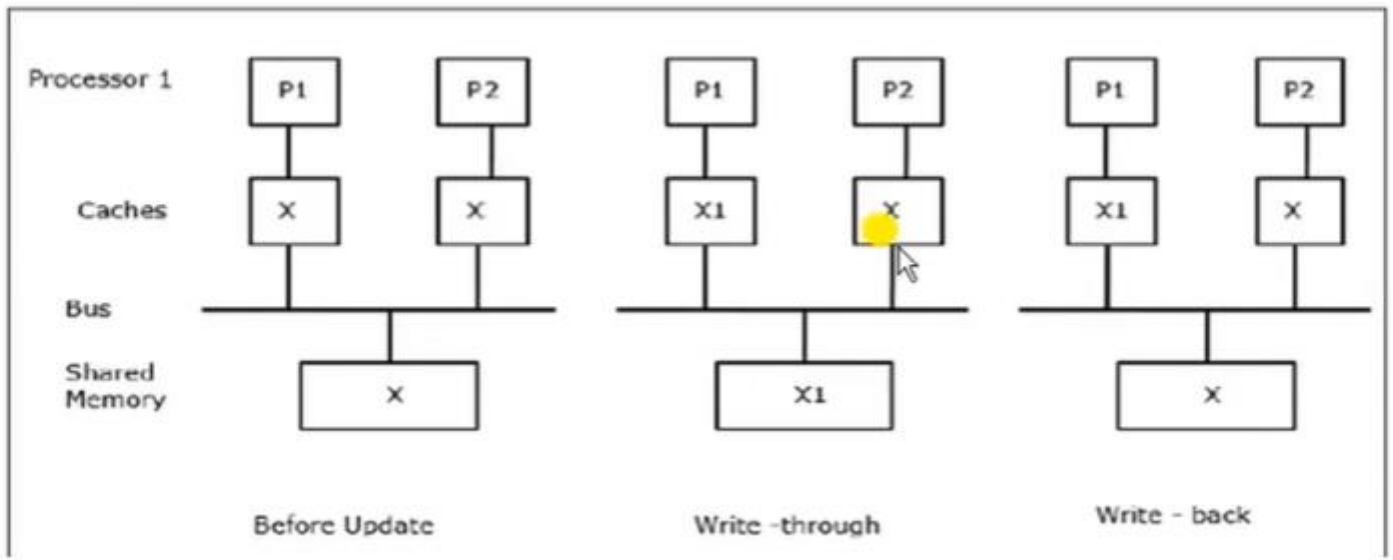
- ***Write through*** : In the ***Write through policy,*** all data written to the cache is also written to memory at the same time (ensuring that main memory is always valid).
- ***Write back*** : In the ***Write back policy,*** when data is written to a cache, a *dirty bit* is set for the affected block. The modified block is written to memory only when the block is replaced.





Example

- Let **X** be an element of **shared data** which has been referenced by **two processors, P1 and P2.** In the beginning, three copies of X are consistent.
- If the **processor P1 writes a new data X1 into the cache,**
  - By using **write-through policy**, the same copy will be written immediately into the shared memory. In this case, inconsistency occurs between cache memory and the main memory.
  - When a **write-back policy** is used, the main memory will be updated when the modified data in the cache is replaced or invalidated.



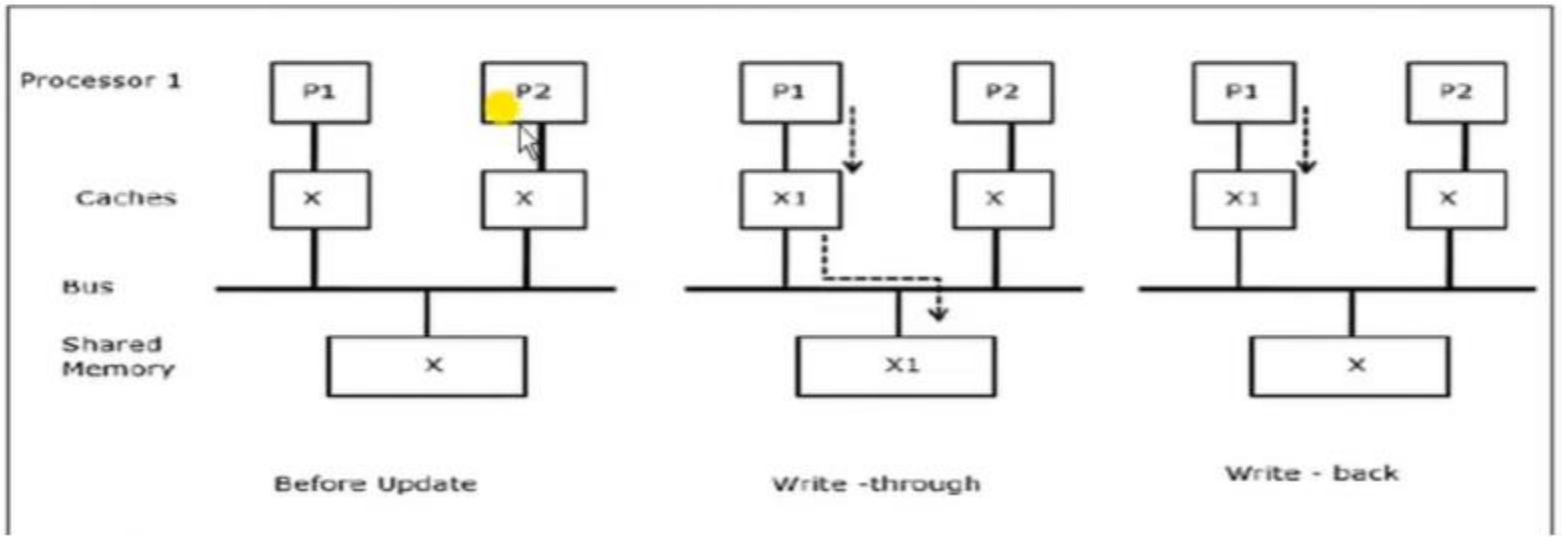
Cause of Cache inconsistencies

In general, there are three sources of inconsistency problem –

- Sharing of writable data
- Process migration
- I/O Operation

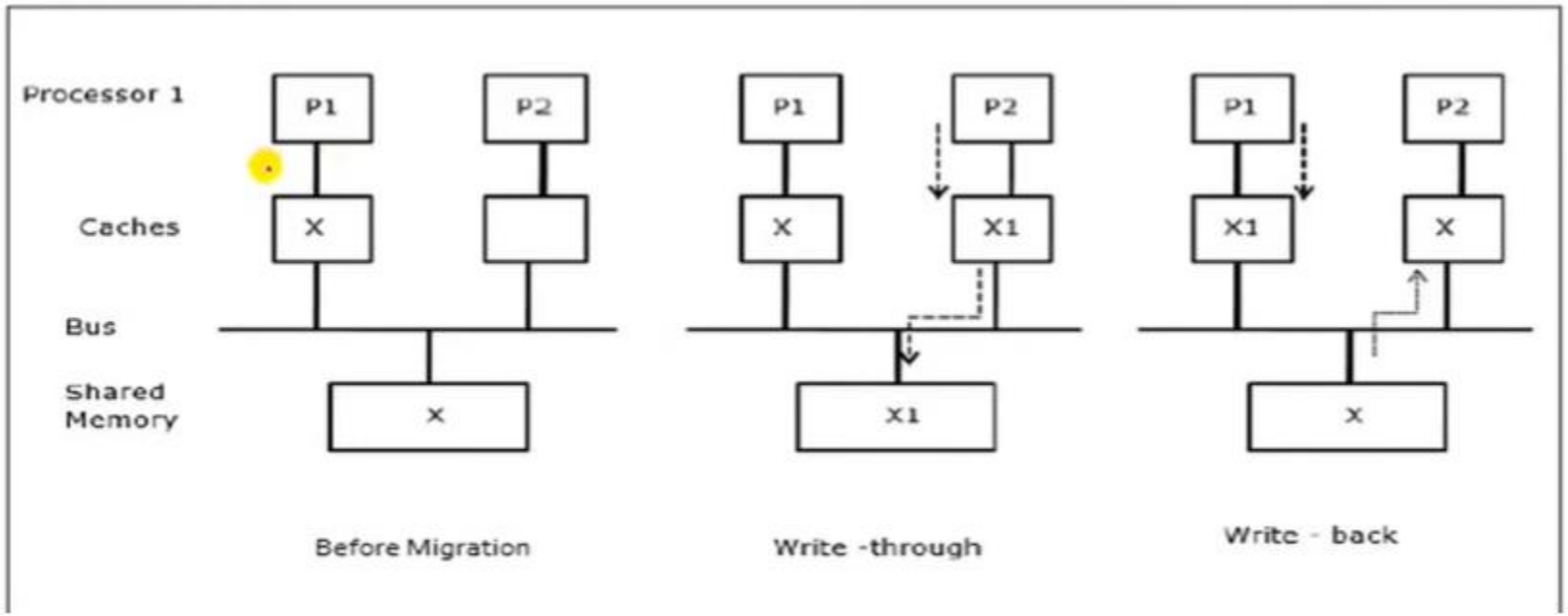
1. Inconsistency due to Data Sharing

- When two processors P1 and P2 have same data element X in their local caches and one processor P1 modifies X (to X1), then the **processor's cache becomes inconsistent with other processor's cache and the shared memory.**
  - With the ***write-through cache***, the shared memory copy will be made consistent by immediately writing into the shared memory, but the other processor still has an inconsistent value (X).
  - With the ***write-back cache***, inconsistency may also occur, in which the shared memory copy will be updated eventually, when the block containing X (actually X1) is replaced or invalidated.



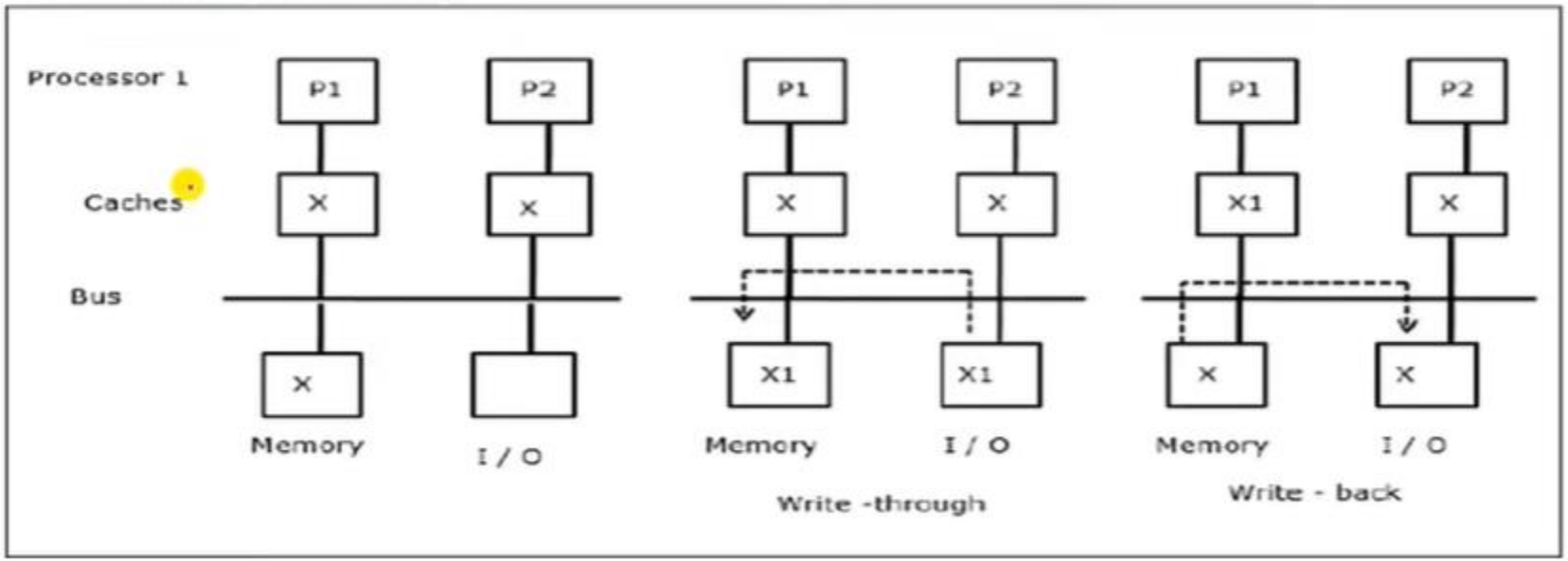
Inconsistency after Process Migration…

- In the first stage, cache of P1 contains X, whereas P2 does not have anything.
- With write-through cache, if a process on P2 modifies X (to X1) and then migrates to P1. After migration, the P1 cache contains X, which is outdated value so inconsistency occur.
- With write-back cache, if a process on P1 modifies X (to X1) and then migrates to P2. After migration, the P2 cache and shared memory both contains data X, which is outdated value, so inconsistency also occur.



3. Inconsistency caused by I/O

- **Data movement from an I/O device to a shared primary memory usually does not cause cached copies of data to be updated** i.e. I/O operation by-passes the caches
- As a result, when the I/O processor loads (inputs) a new data X1 into the main memory (by-passing the writhe-through cache), inconsistency occurs between caches and the shared memory
- Likewise, when outputting a data from the shared memory to I/O device (by-passing the writhe-back cache), also creates inconsistency.



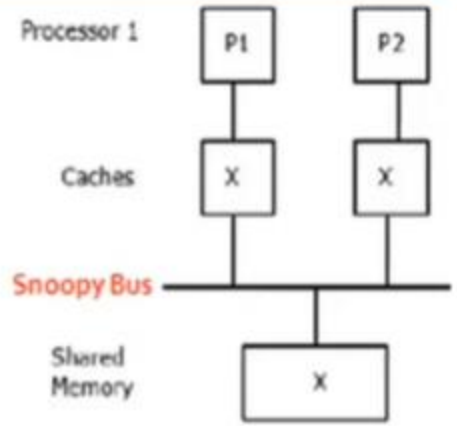
Cache Coherence Solutions

(Cache Coherence Protocols)

- **Snoopy-Bus Protocol**
  - Used for bus-based multiprocessor systems (UMA Machines)
  - In this protocol, all the processors observe memory transactions and take proper action to invalidate or update the local cache content if needed.
- **Directory Based Protocol**
  - Used in scalable multiprocessor systems (NUMA Machines) interconnected using crossbar switch
  - In this protocol, cache directories are used to keep a record on where copies of cache blocks reside.

Snoopy-Bus Protocol

- Used for bus-based multiprocessor systems (UMA Machines)
- Transactions on bus are visible to all processors.
- In **Snoopy-Bus protocol**, *each processor's* ***cache controller monitors or snoops on the bus for memory transactions*** *and* ***take*** *proper* ***action to invalidate or update the local cache content*** *if needed*
  - A bus allows all processors in the system to observe ongoing transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate or update the local copy. It is called as snoopy protocol because each cache snoops on the transaction of other caches
- Shortcoming : not scalable
- More scalable solution: 'directory based' coherence schemes



Two Basic Protocols:

Using private caches associated with processors tied to a common bus, two approaches are used to maintain cache consistency

1. Write-invalidate:

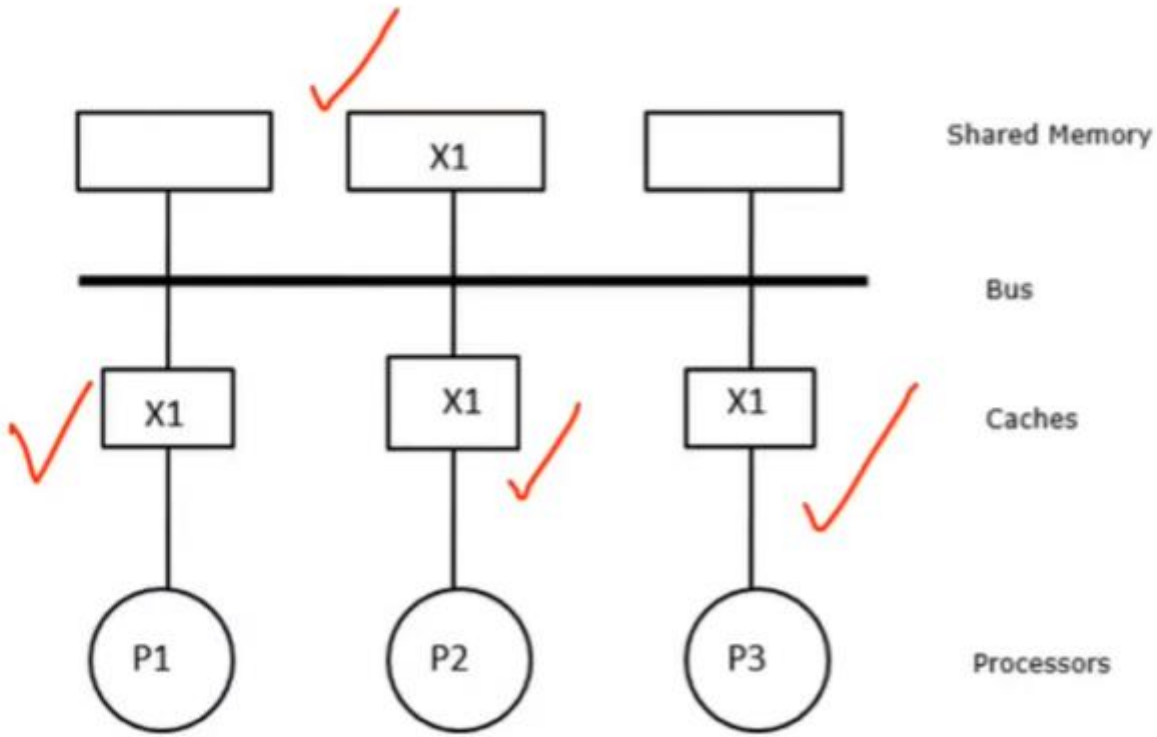
When a local cache copy is modified, **Write-invalidate policy** **it invalidates all remote copies of cache** (invalidated items are sometimes called "dirty")

2. Write-update (Write-broadcast):

When a local cache copy is updated, **Write-update policy** **broadcast a modified value of a data object to all other caches at the time of modification**

write-update protocol

- ***The write-update protocol demands,***
  - The new modified content X1 be **broadcast (update)** to all cache copies via the bus.



(c) After write-update operation by P1

- The memory copy is also updated if **write-through caches** are used.
- In using **write-back caches,** the memory copy is updated later at block replacement time

Directory Based Protocols

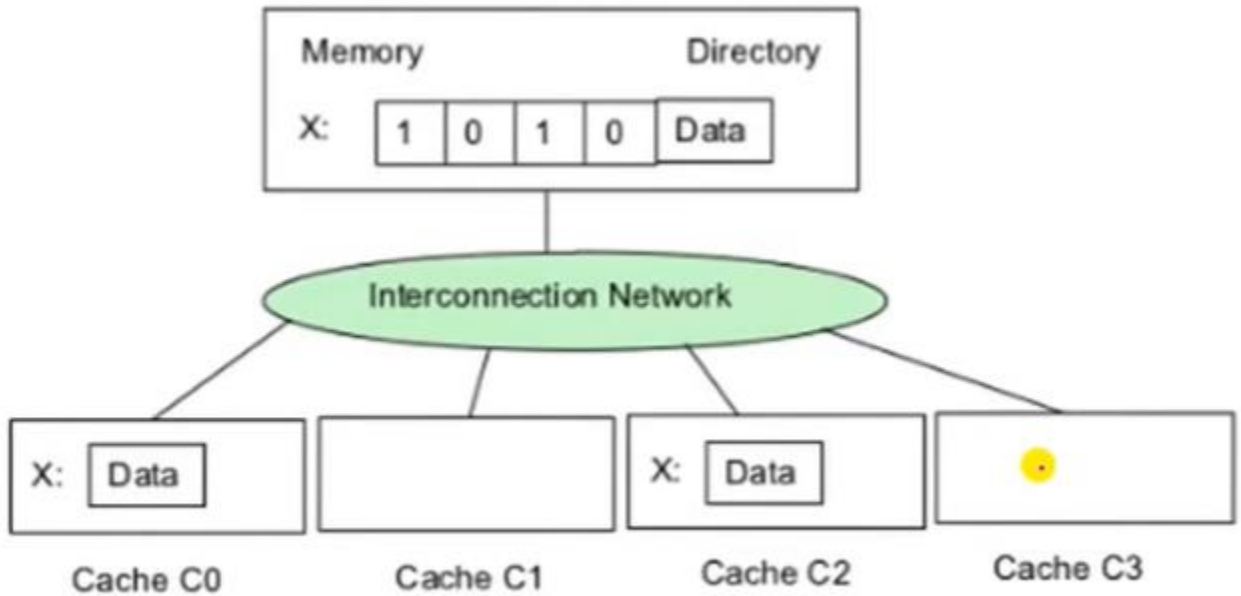
- Used in scalable multiprocessor systems (NUMA Machines) interconnected using crossbar switch
- ***Directory based cache coherence protocol,*** uses **cache directories** **to keep track of the status of all cache blocks**
  - The **directory** contains global state information about the contents of various local cache
- The directory acts as a filter where the processors ask permission to load an entry from the primary memory to its cache memory.
- If an entry is changed, the directory either updates it or invalidates the other caches with that entry.
- Its actually existed BEFORE Snoop-based schemes. Snoopy-bus schemes are not scalable. So Directory based schemes can be solution.

Directory Based Protocols

- Protocol Categorization:
  1. Full Map Directories
  2. Limited Directories
  3. Chained Directories

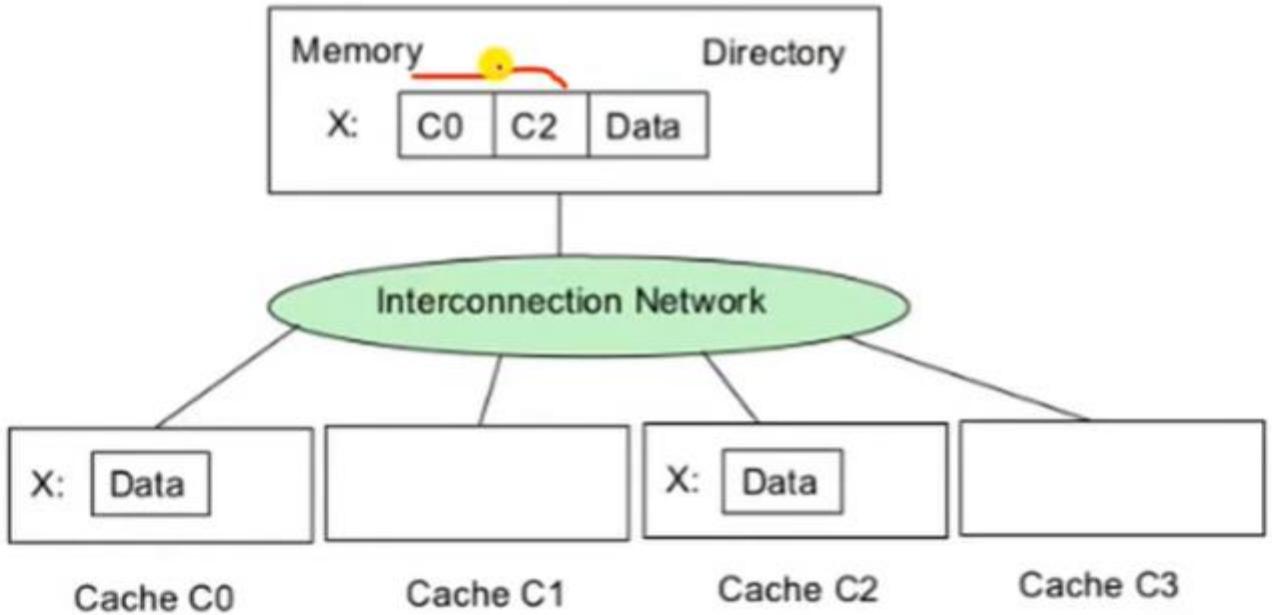
1. Full-map Directories

- Each directory entry contains N pointers, where N is the number of processors
- There could be N cached copies of a particular block shared by all processors
- For every memory block, an N bit vector is maintained, where N equals the number of processors in the shared memory system. Each bit in the vector corresponds to one processor



2. Limited Directories

- Fixed number of pointers per directory entry regardless of the number of processors.
- Restricting the number of simultaneously cached copies of any block should solve the directory size problem that might exist in full-map directories.



3. Chained Directories

- Chained directories emulate full-map by distributing the directory among the caches.
- Solving the directory size problem without restricting the number of shared block copies.
- Chained directories keep track of shared copies of a particular block by maintaining a chain of directory pointers