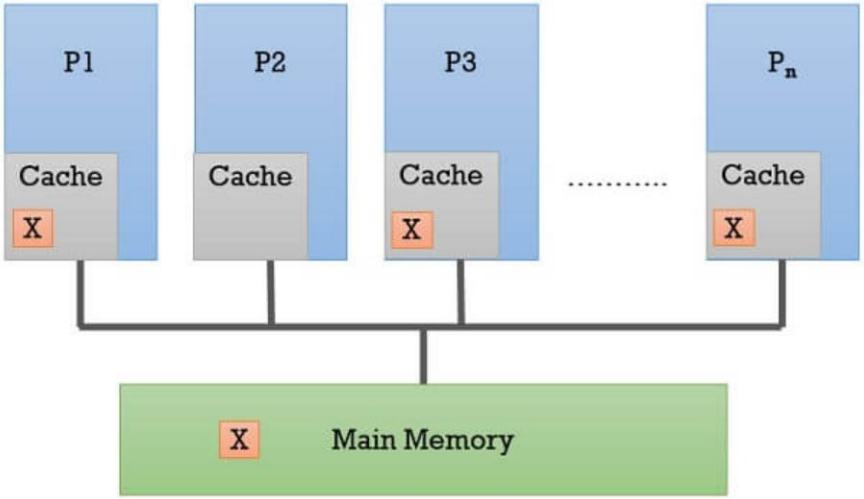
CACHE COHERENCE

Cache Coherence

- What is Cache Coherence Problem?
  - In a multiprocessor environment, all the processors in the system share the **main memory** via a **bus.**
  - For better performance, each processor implements its own **cache.** Processors may share the same data block by keeping a copy of this data block in their cache.



- In case, the processors P1 modifies the copy of shared memory block X present in its cache. It would result in **data inconsistency**. As the processor **P1** will have the **modified copy** of shared memory block i.e. X1. But, the **main memory** and other processor's **cache** will have the **old copy** of shared memory block. And this problem is the **cache coherence problem**.

Write Policies

Write Back: Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.

Write Through: All write operations are made to main memory as well as to cache, ensuring that main memory is always valid

Write back policy can result in inconsistency. If two caches contain the same line, and the line is updated, other cache will unknowingly have invalid value. Subsequent reads to that invalid line produce invalid results

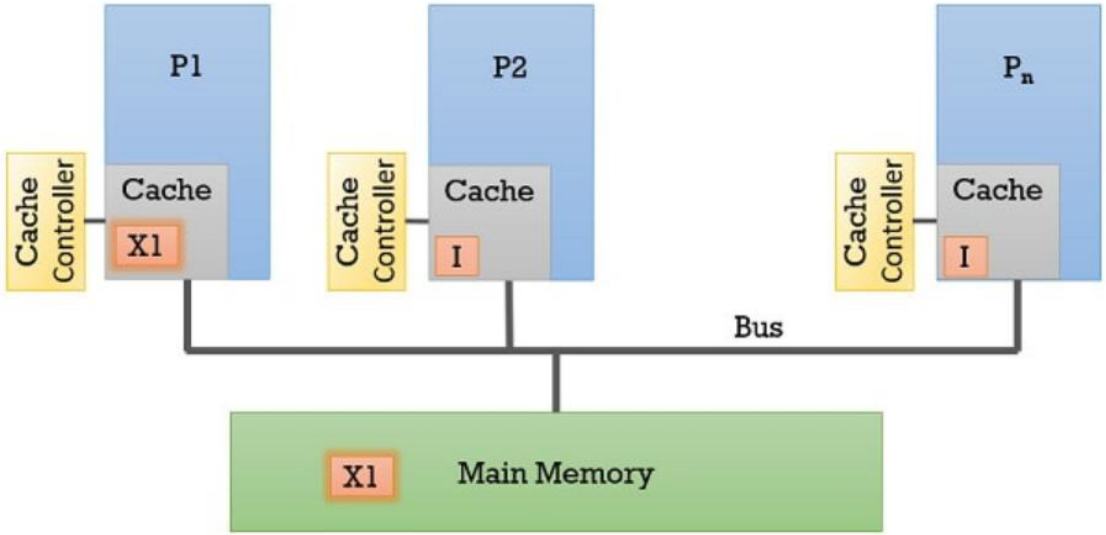
Even with Write through policy, inconsistancy can occur unless other caches monitor the memory traffic or receive some direct notification of the update

- This cache coherence problem can be sorted using protocols:
  - Snoopy Protocol
  - Directory Based Cache Coherence Protocol
  - MESI Protocol

Snoopy Protocol

- In the multiprocessor environment, all the processors are connected to memory modules via a **single bus**. The transaction between the processors and the memory module i.e. read, write, invalidate request for the data block occurs via bus.
- If we implement the **cache controller** to every processor's cache in the system, it will **snoop** all the **transaction over the bus** and perform the appropriate action. So, we can say that the Snoopy protocol is the **hardware solution** to cache coherence problem.
- It is used for small multiprocessor environment as the large multiprocessors are connected via the interconnection network.

Snoopy Protocol



- Consider a scenario from write-back, if a processor has just modified a data block in its cache, and is a current owner of the block.
- Now, if processor P1 wishes to modify the same data block that has been modified. P1 would broadcast the invalidation request on the bus and becomes the owner for that data block and modifies the data block. The other processors who have the copy of the same data block **snoop** the bus and invalidate their copy of the data block (I). It updates memory using write-back protocol.

- Snoopy protocols distribute the responsibility for maintaining cache coherence among all the cache controllers in a multiprocessor. A cache must recognize when a line it holds is shared with other caches. When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism. Each cache controller is able to "snoop" on the network to observe these broadcasted notifications and react accordingly.
- Two basic approaches to the snoopy protocol:
  - Write Invalidate
  - Write update(or Write Broadcast)

Write Invalidate:

Invalidating the affected copies of shared data in other processors cache.

- When a processor modifies any data block in its cache it also **updates the main memory** with the modified copy of the same data block.
- The processor modifying the data block sends the broadcast requests to other processors in the system to **invalidate the copies of the same data block** in their caches.
- By invalidating the line in other caches, Processor makes the cache exclusive. Once the line is exclusive the owning processor can make local writes until some other processor requires the same line

Write Update:
Updating the affected copies of shared data in other processors cache.

- When a processor modifies a data block in its cache and it also **updates the main memory** with the new copy of the same data block. The other processors in the system also have the copy of the data block that has been modified and now they carry the invalid value.
- To update the copies in other caches. The processor that modifies the data block, **broadcast the modified data** to all the processors in the system. On receiving the broadcast data, the processor verifies whether it has the same data block in its cache if present, then it modifies the content of that data block else discard the broadcasted data.

Directory Based Protocols

- Directory protocols collect and maintain information about where copies of lines reside.
- There is a Centralized controller that is part of main memory controller and a directory that is stored in main memory.
- The directory contains global state information about the contents of the various local caches.
- When an individual cache controller makes a request, the centralized controller checks and issues necessary commands for data transfer between memory and caches. It is also responsible for keeping the state information up to date; therefore, every local action that can affect the global state of line must be reported to the central controller.

- The controller maintains information about which processor have a copy of which line.
- Before a processor can write to a local copy of a line, it must request exclusive access to the line from the controller. Before granting this exclusive excess, the controller sends a message to all processors with a cached copy of this line, forcing each processor to invalidate its copy. After receiving acknowledgements back from each processor, the controller grants exclusive access to the requesting processor.
- When another processor tries to read a line that is exclusive granted to another processor, it will send miss notification to the controller. The controller then issues a command to the processor holding that line that requires the processor to do write back to main memory. The line may now be shared for reading by requesting processor

Drawback

- Central Bottleneck
- Overhead of communication
- Size of directory

Types

- Full Map Directories
  - Each directory entry contains N pointers, where N is the number of processors.
  - For every memory block , an N bit vector is maintained, where N equals the number of processors. Each bit in vector corresponds to one processor.

- Chained Directory
  - Distribute the directory among the caches
  - Chained directory keep track of shared copies of a particular block by maintaining a chain of directory pointers

MESI Protocol

- MESI is a cache coherence protocol that assures data consistency on a **symmetric multiprocessor (SMP)**. The word MESI defines the four states of the data block in the caches of the processors of the multiprocessing system.
- **Modify (M):** The data block in a cache is **modified** and processor modifying the data block is the owner of that data block. This copy of the data block is not available with any other caches in the system.The **main memory** copy for the same data block does **not contain the modified value** of the data block. If the processor wants to modify it again, it doesn't need to broadcast this request over the bus again.
- **Exclusive (E):** The line in the cache is the same as that in the main memory and is not present in any other cache
- **Shared(S):** The line in the cache is the same as that in main memory and may be present in another cache
- **Invalid (I):** The line in the cache does not contain valid data

MESI cache line states

	M Modified	E Exclusive	S Shared	I Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is…	Out of date	Valid	Valid	
Copies exist in other caches?	No	No	May be	May be
A write to this line	Does not go to bus	Does not go to bus	Goes to bus and updates cache	Goes directly to bus

Each line of the cache has its own state bits and therfore its own realization of the state diagram.

MESI state Transition

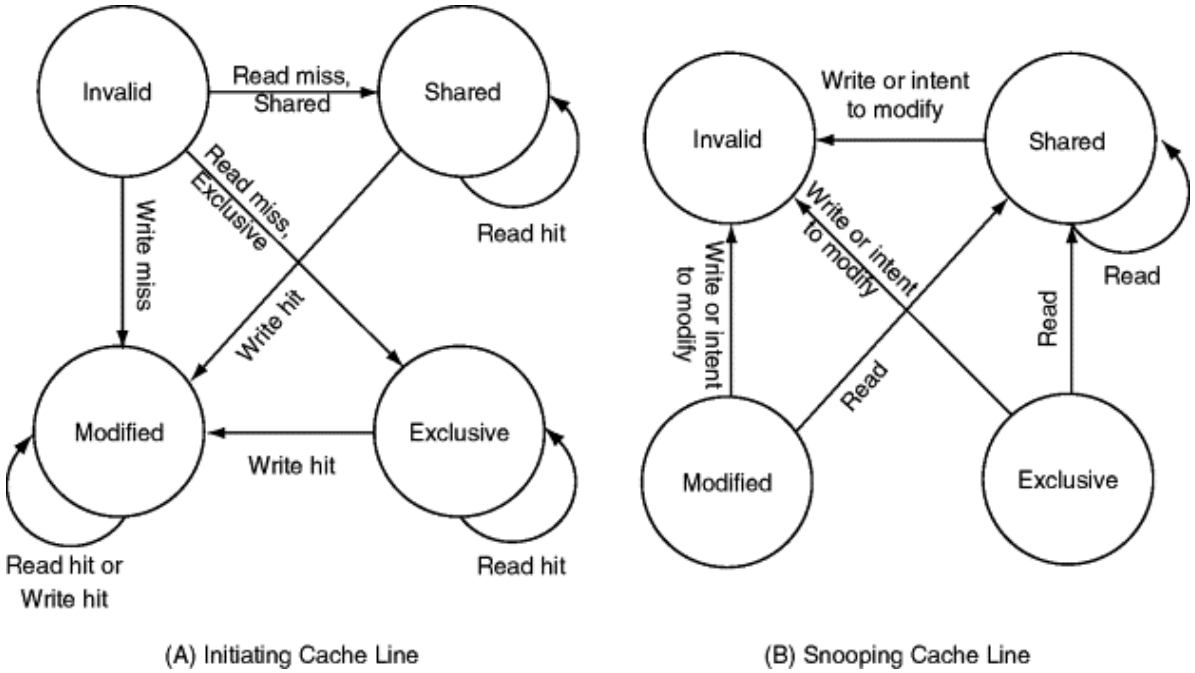


Fig A shows the transition that occur due to actions initiated by the processor attached to this Cache.

Fig B shows the transitionns that occur due to events that are snooped on the common bus

- At any time a cache line is in a single state. If the next event is from the attached processor, then transition is dictated by Fig A and if the next event is from the bus the transition is dictated by Fig B.

- **Read HIT:** When a read hit occurs on a line currently in the local cache, the processor simply reads the required item. There is no state change.

- **Read Miss:** When read miss occurs in the local cache, the processor initiates a memory read. The processor inserts signal on the bus that alerts all other processors/cache units to snoop the transaction. There are number of possible outcomes:
  - If another cache has a clean copy of the data in the exclusive state, it returns a signal indicating that it shares this data. The responding processor then transitions the state of its copy from exclusive to shared and the initiating processor reads the line from main memory and transitions the line of its cache from invalid to shared.
  - If another cache has modified copy of the line, then that cache blocks the memory read and provides the line to the requesting cache over the shared bus. The responding cache changes its line from modified to shared. The line sent to the requesting cache is also received and processed by the memory controller, which stores the block in memory.
  - If no other cache has a copy of the line, then no signals are returned. The initiating processor reads the line and transitions the line in its cache from invalid to exclusive.

- Write Miss: When write Miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. For this purpose, the processor issues a signal on the bus that means read-with-intent-to-modify. When the line is loaded, it is immidiately marked modified. With respect to other caches , two possible scenarios precede the loading of the line data
  - Case 1: Some other cache may have a modified copy of this line. In this case, the altered processor signals the initiating processor that another processor has modified copy of the line. The initiating processor surrenders the bus and waits. The other processor gains access to the bus, writes the modified cache line back to main memory and transitions the state of the cache line to invalid(because the initiating processor is gong to modify this line). Subsequently the initiating processor will again issue a signal to the bus of read-with-intent-to-modify and then read the line from main memory, modify the line in the cache and mark the line in the modified state.
  - Case 2: No other cache has a modified copy of the requested line. In this case no signal is returned, and the initiating processor proceeds to read the line and modify it. Meanwhile, if one or more caches have a clean copy of the line in the shared state, each cache invalidates its copy of the line

- **Write Hit:** When Hit occurs on the line currently in the local cache, the effect depends on the current state of that line in the local cache
  - Shared: Before performing the update, the processor must gain exclusive ownership of the line. The processor signals its intent on the bus. Each processor that has a shared copy of the line in its cache, thansitions from shared to invalid. The initiating processor the performs the update and transitions from shared to modified)
  - Exclusive: processor Performs update and transitions from Exclusive to modified
  - Modified: Simply performs the update

- William Stallings, Computer Organization and Architecture, Pearson, 9th Ed
- https://binaryterms.com/cache-coherence.html

Thankyou