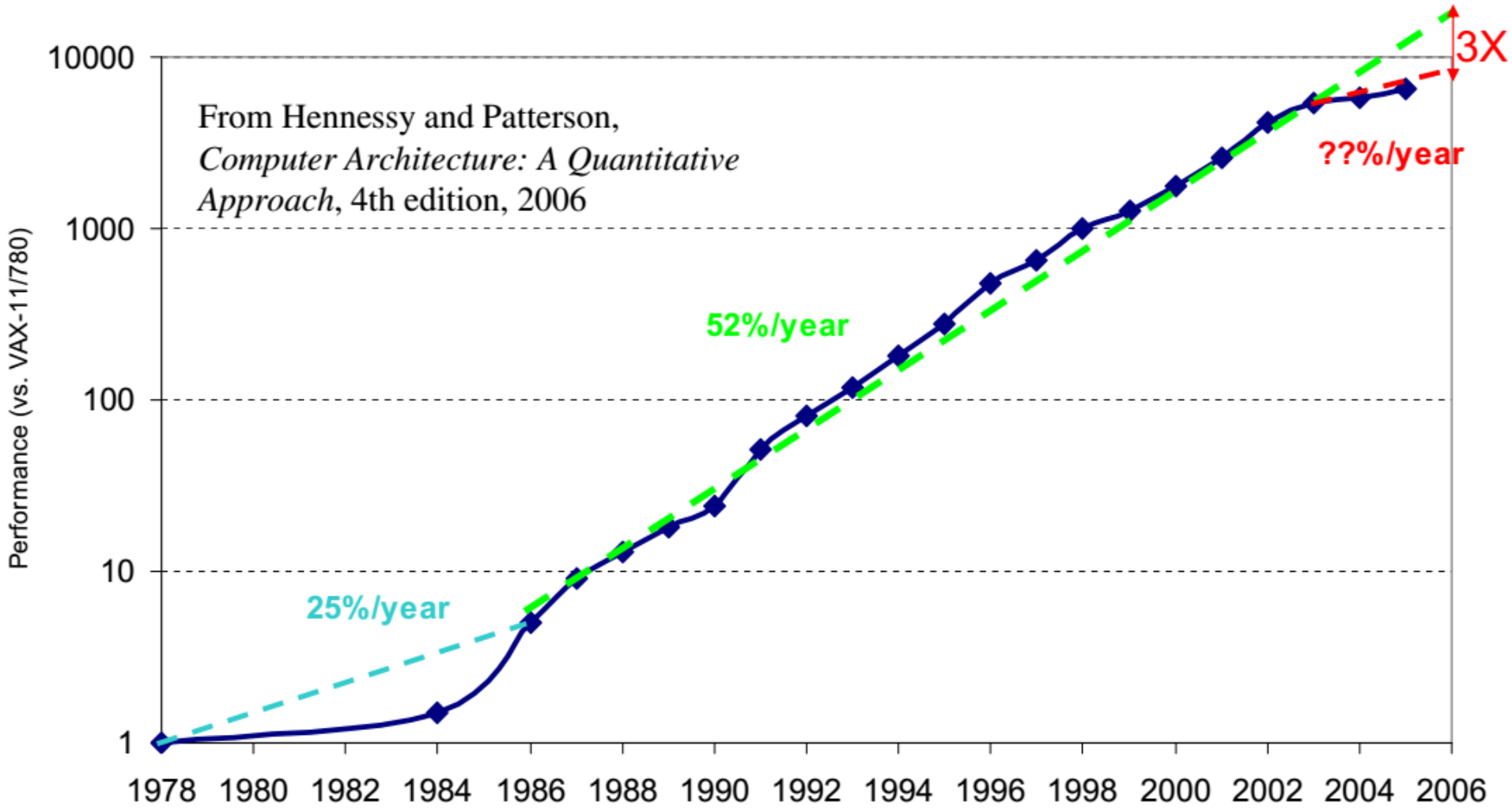
Multiprocessors and Thread-Level Parallelism

Uniprocessor Performance (SPECint)



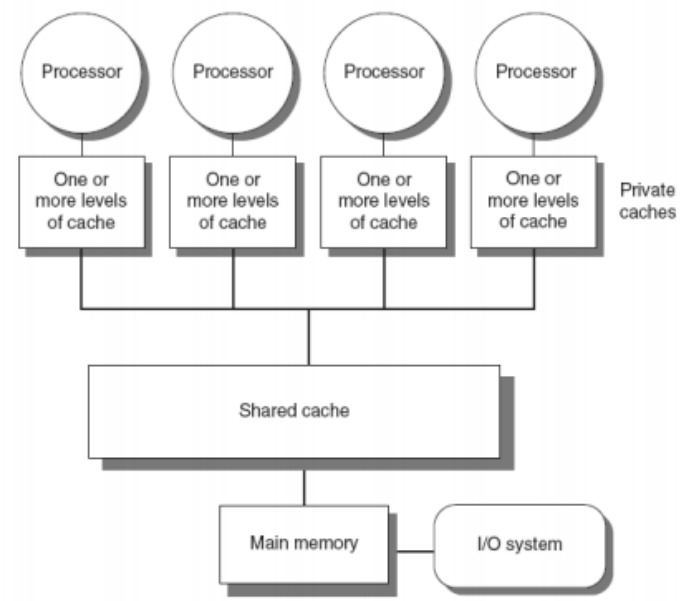
- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

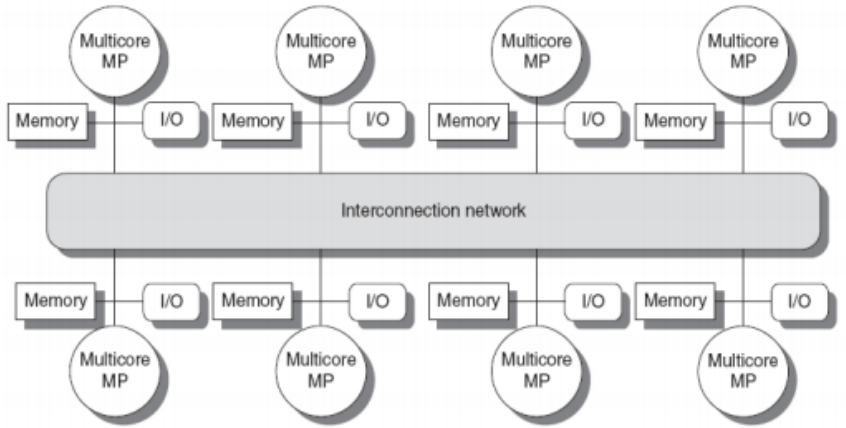
Introduction

- Thread-Level parallelism
  - Have multiple program counters
  - Uses MIMD model
  - Targeted for tightly-coupled shared-memory multiprocessors
- For *n* processors, need *n* threads
- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

Types

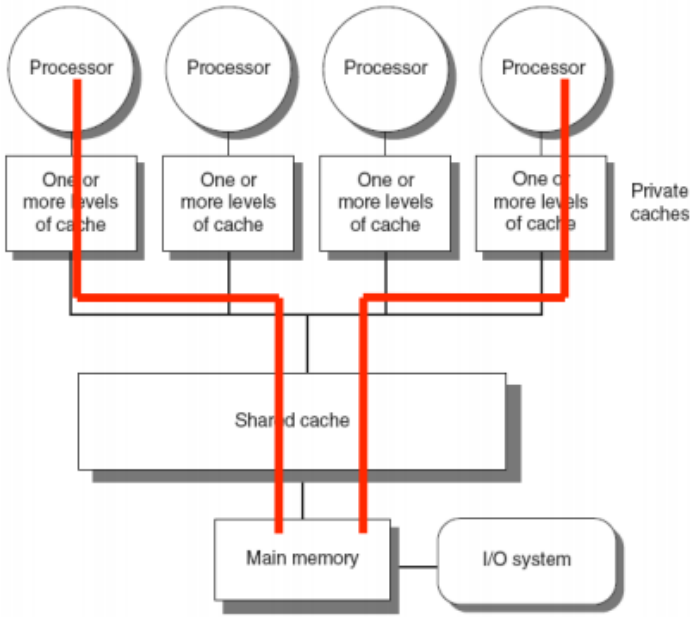
- Symmetric multiprocessors (SMP)
  - Small number of cores
  - Share single memory with uniform memory latency
- Distributed shared memory (DSM)
  - Memory distributed among processors
  - Non-uniform memory access/ latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks

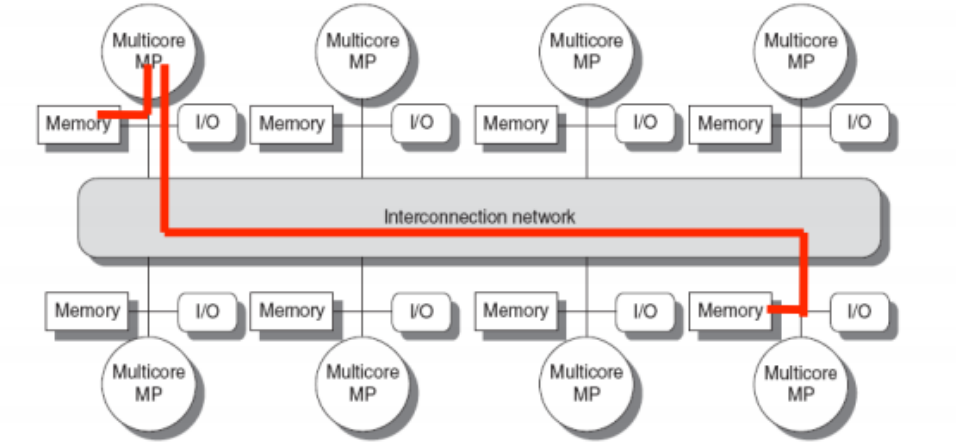




SMP and DSM

- Shared memory:
  - Communication among threads through a shared address space
  - A memory reference can be made by any processor to any to any memory location.





Challenges of Parallel Processing

- First challenge is % of program inherently sequential

- Suppose 80X speedup from 100 processors. What fraction of original program can be sequential?

  a. 10%

  b. 5%

  c. 1%

  d. <1%

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$80 = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

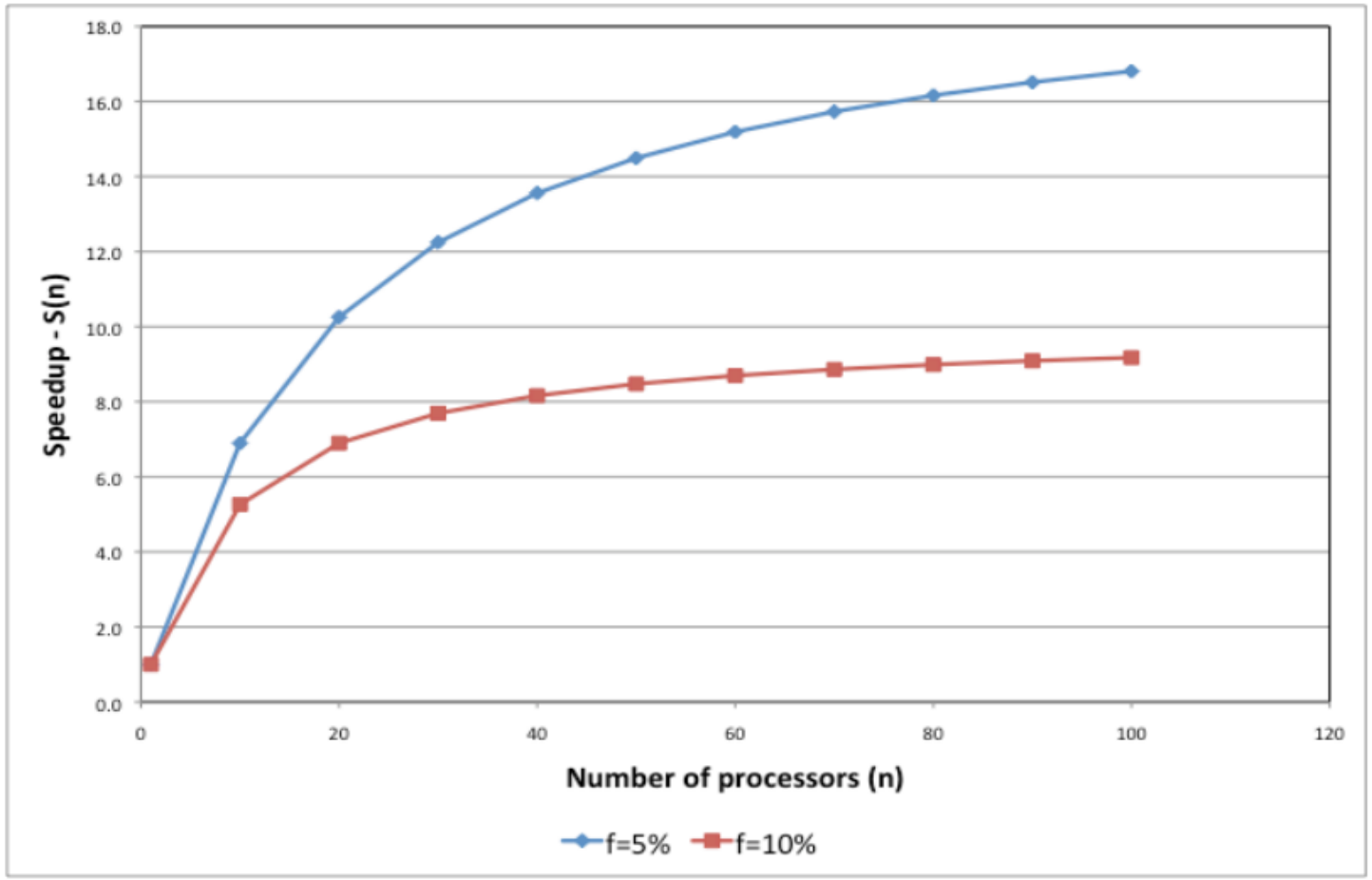
$$80 \times \left[\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}\right] = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

Challenges of Parallel Processing

- How does the speedup vary with n and f?



Challenges of Parallel Processing: Large Latency of Remote Access

- An application runs on a 32-processor MP that has a 200 ns time to handle a reference to remote memory. Assume that all references (except hose involving communication) hit the cache. The processor clock rate is 3.3 GHz and CPI (cycles per instruction) is 0.5 if all instructions hit the local cache. How much faster is the MP if there is no communication vs. the case in which 0.2% of the instructions involve a remote reference?

$$CPI = 0.5 + 0.2\% \times \text{Remote Request Cost}$$

$$\text{Remote Request Cost} = \frac{\text{Remote Access Cost}}{\text{Cycle time}} = \frac{200\text{ns}}{1/3.3\text{ ns}} = 660 \text{ cycles}$$

$$\text{Effective } CPI = 0.5 + 0.2\% \times 660 = 1.82$$

- MP with no remote access is 1.82/0.5=3.64 times faster.

Centralized Shared Memory Architectures

- SMPs: both shared and private data can be cached.
- Shared data provides a mechanism for processors to communicate through reads and writes to shared memory.
- The effect of caching private data on program behavior is the same as that of a uniprocessor because no other processor access these data.
- The value of shared data may be replicated in the multiple caches:
  - + reduction in cache contention
  - - cache coherence!

Cache Coherence

For higher performance in a multiprocessor system, each processor will usually have its own cache. Cache coherence refers to the problem of keeping the data in these caches consistent. The main problem is dealing with writes by a processor.

There are two general strategies for dealing with writes to a cache:

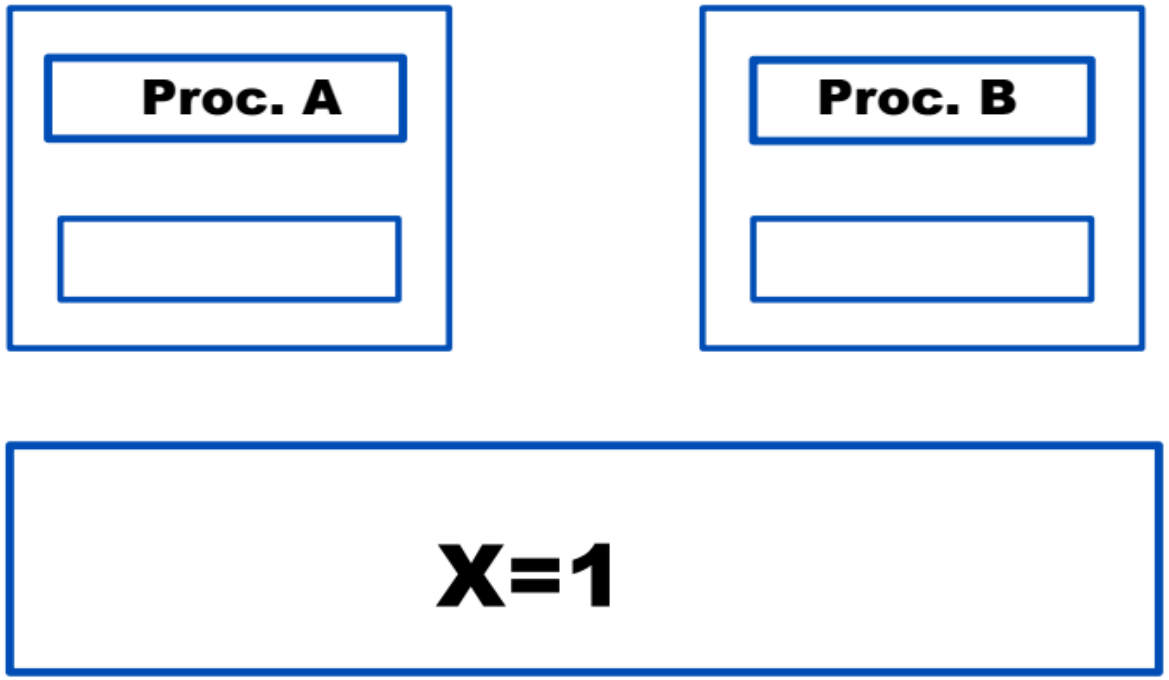
Write-through - all data written to the cache is also written to memory at the same time.
Write-back - when data is written to a cache, a dirty bit is set for the affected block. The modified block is written to memory only when the block is replaced.

Write-through caches are simpler, and they automatically deal with the cache coherence problem, but they increase bus traffic significantly. Write-back caches are more common where higher performance is desired.

Cache coherence is a concern in a multicore environment because of distributed L1 and L2 caches. Since each core has its own cache, the copy of the data in that cache may not always be the most up-to-date version. For example, imagine a dual-core processor where each core brought a block of memory into its private cache, and then one core writes a value to a specific location. When the second core attempts to read that value from its cache, it will not have the most recent version unless its cache entry is invalidated and a cache miss occurs. This cache miss forces the second core's cache entry to be updated. If this coherence policy was not in place, the wrong data would be read and invalid results would be produced, possibly crashing the program or the entire computer.

Cache Coherence

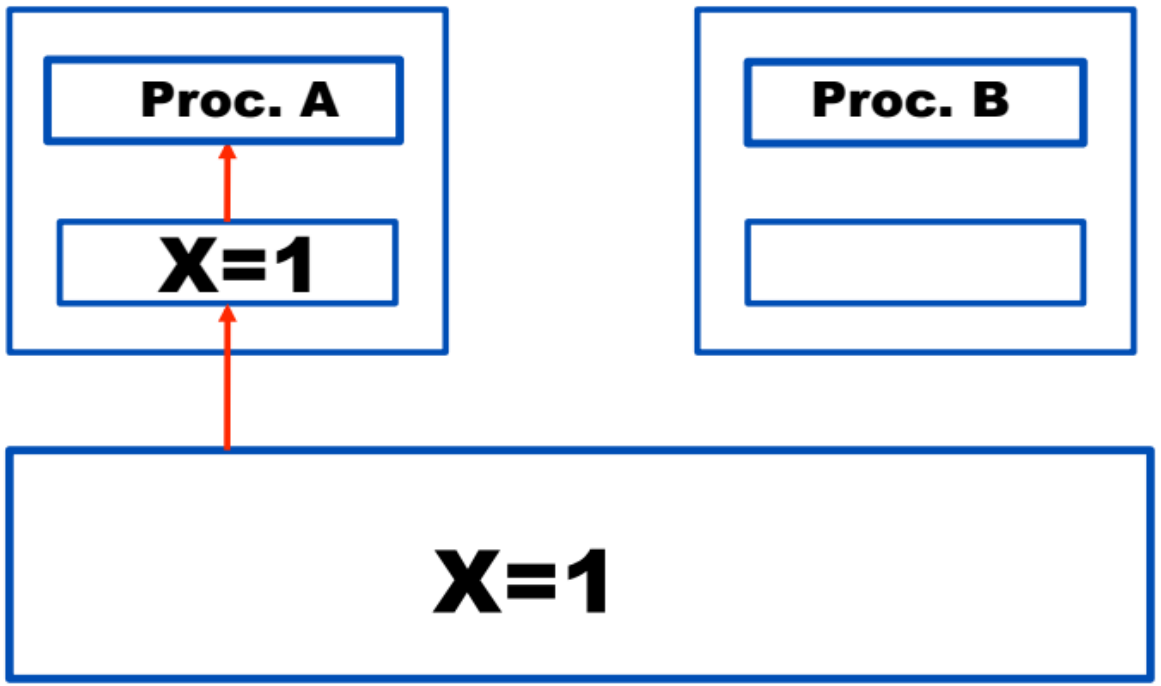
- Processors may see different values through their caches:



- Processor A reads X (cache miss)

Cache Coherence

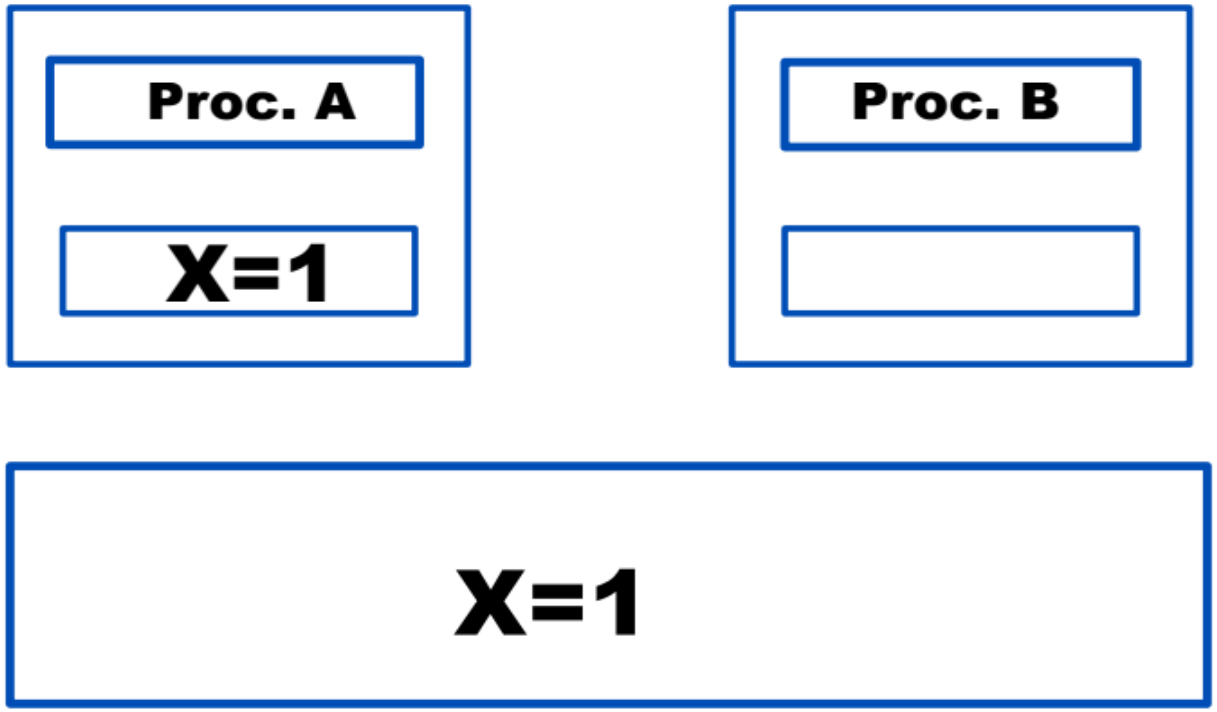
- Processors may see different values through their caches:



- Processor A reads X

Cache Coherence

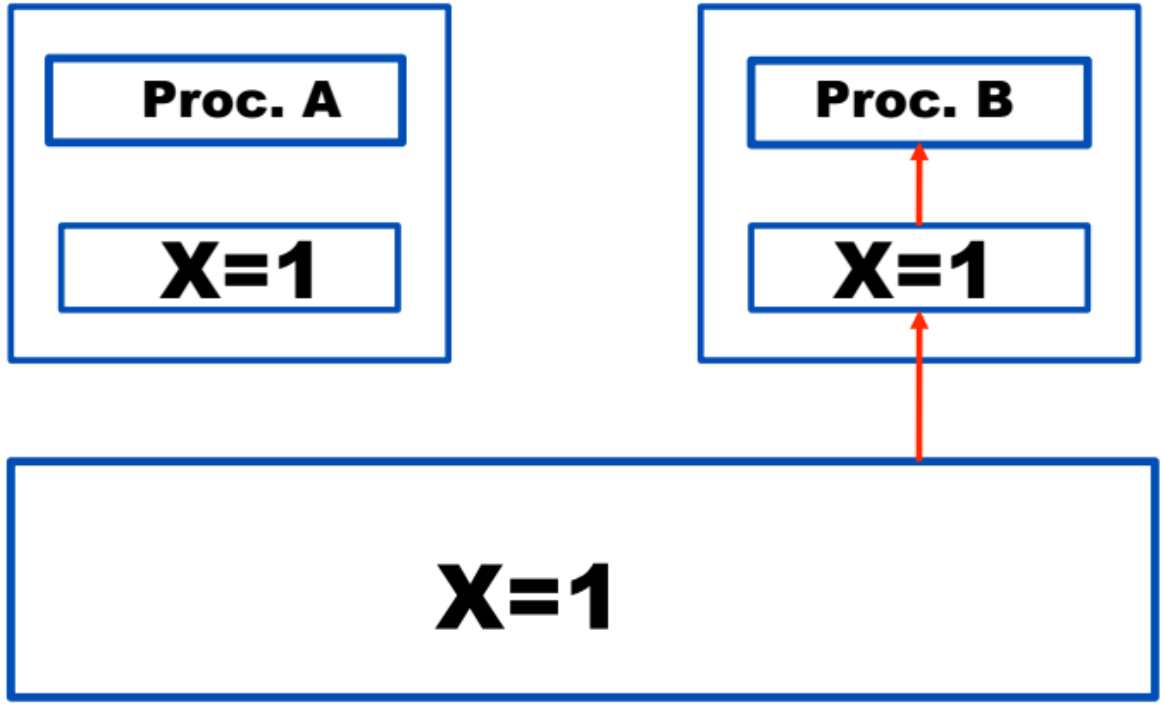
- Processors may see different values through their caches:



- Processor B reads X (cache miss)

Cache Coherence

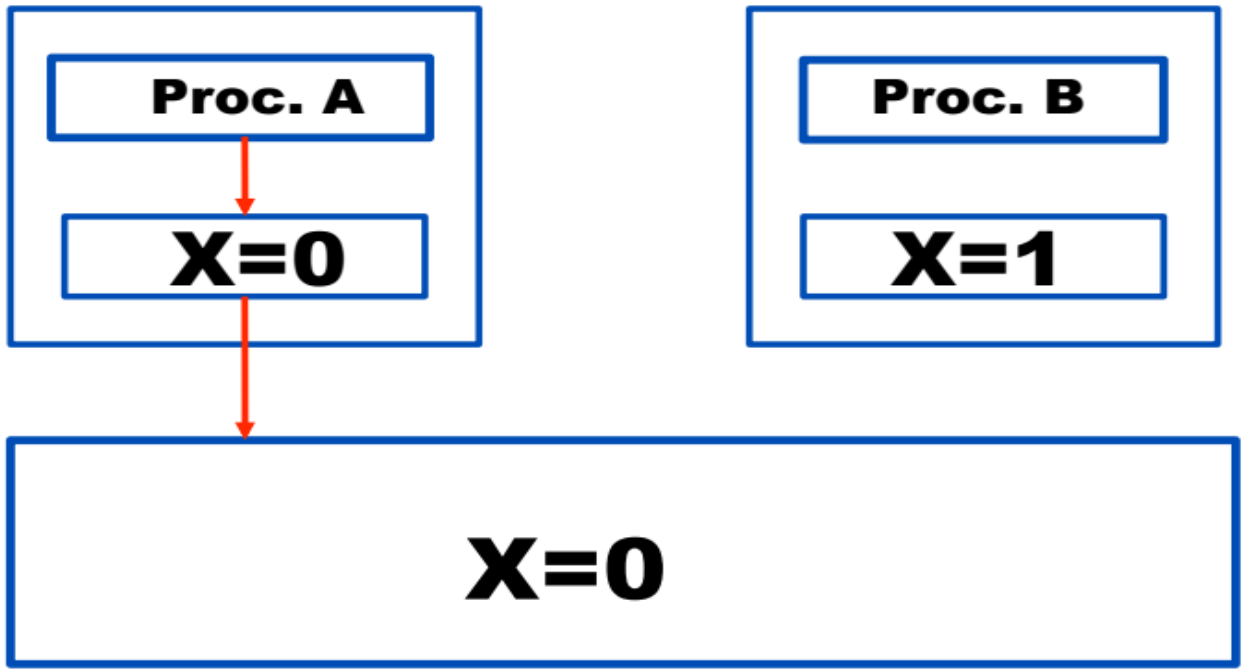
- Processors may see different values through their caches:



- Processor B reads X

Cache Coherence

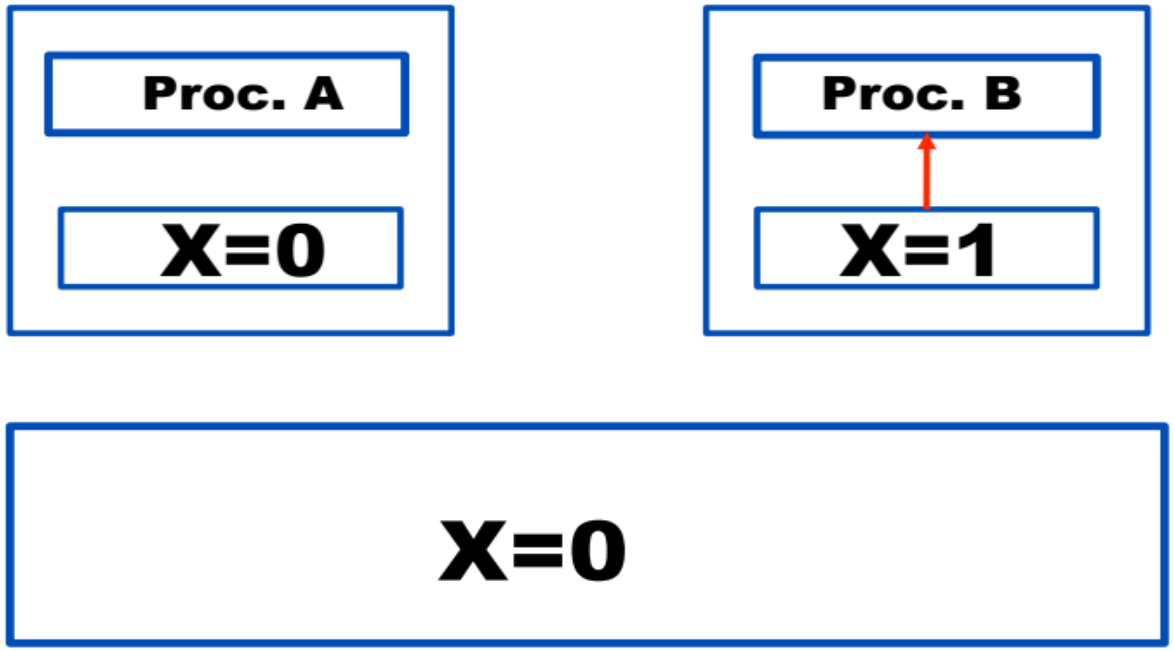
- Processors may see different values through their caches:



- Processor A writes 0 into X (assume write-though cache)

Cache Coherence

- Processors may see different values through their caches:



- Processor B reads X (cache hit) and reads the old value.

Cache Coherence

- If there are no intervening writes by another processor, a read must return the last value written by the same processor.

  Write(P,X,X'), …,Write(P',X,X"), …,X'=Read(P,X)

- A read by another processor must return the value of the latest write if the read and write are sufficiently separated in time.

  Write(P,X,X'), …,Write(P',X,X"), …,X'=Read(P',X)

- Writes to the same location are serialized.

  Write(P,X,X'),Write(P',X,X"),X"=Read(P",X),X'=Read(P",X)

Cache Coherence

- Coherence
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors

- Consistency
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

Enforcing Coherence

- Coherent caches provide:
  - *Migration*: movement of data
  - *Replication*: multiple copies of data

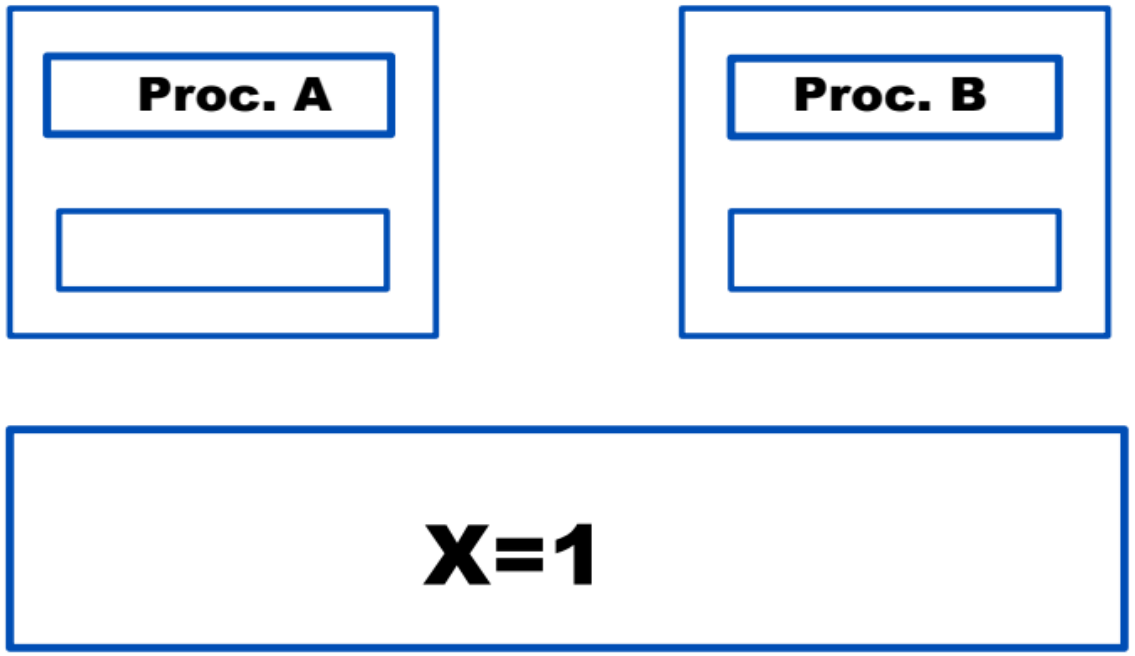
- Cache coherence protocols
  - Directory based
    - Sharing status of each block kept in one location
      - Natural for SMP: shared memory
      - Challenging for DSM
  - Snooping
    - Snooping: cache controllers *snoop* on the broadcast medium to determine if they have a copy of a block being requested.
    - Each core tracks sharing status of each block

Snooping Coherence Protocols

- Write invalidate
  - On write, invalidate all other copies (most common)
  - Use bus to serialize
    - Write cannot complete until bus access is obtained
  - The same item can appear in multiple caches
  - Two caches will never have different values for the same block.
  - Most commonly used in MPs
- Write update (a.k.a. write broadcast)
  - On write, update all copies
  - Consumes considerable bandwidth

Write Invalidate Cache Coherenc

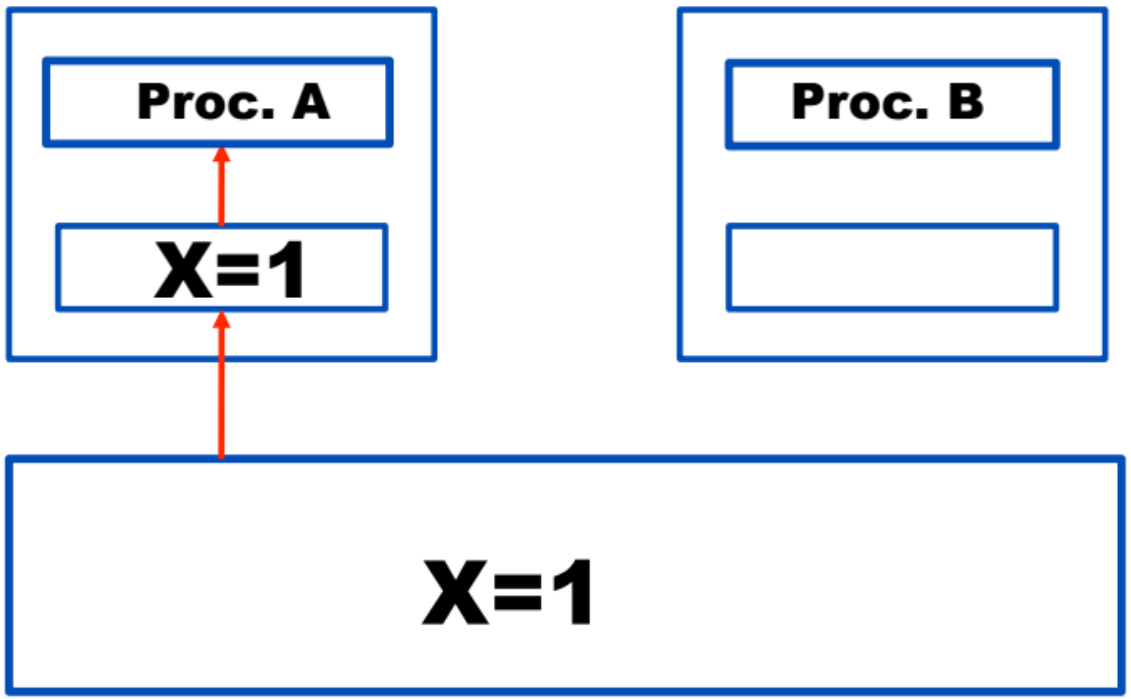
- Invalidate other caches on write:



- Processor A reads X (cache miss)

Write Invalidate Cache Coherence

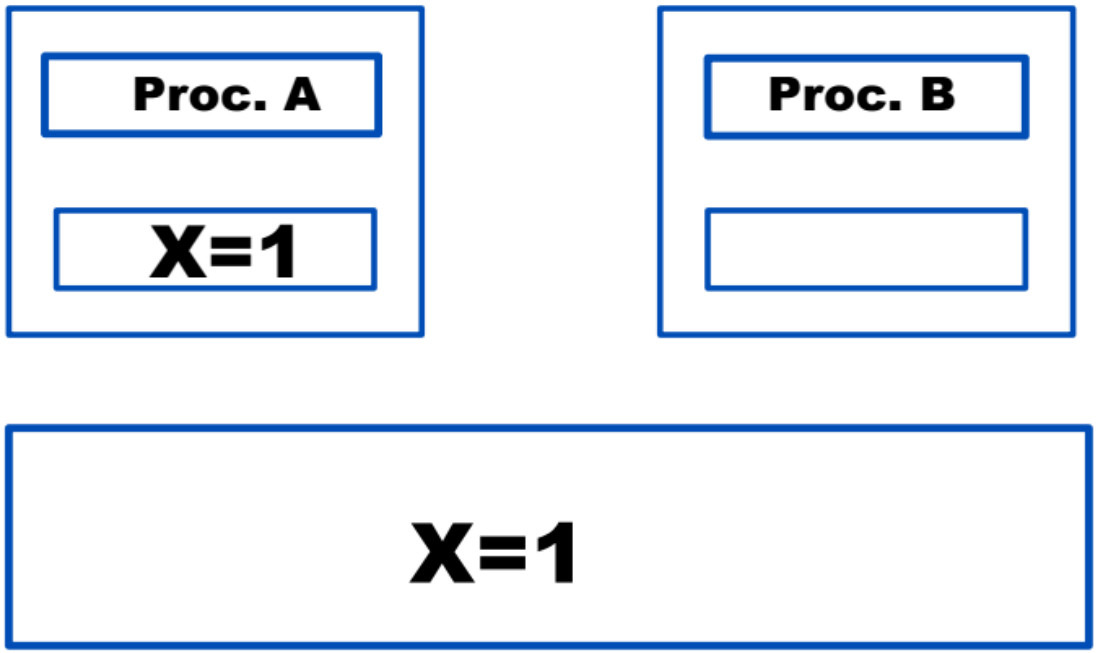
- Invalidate other caches on write:



- Processor A reads X

Write Invalidate Cache Coherence

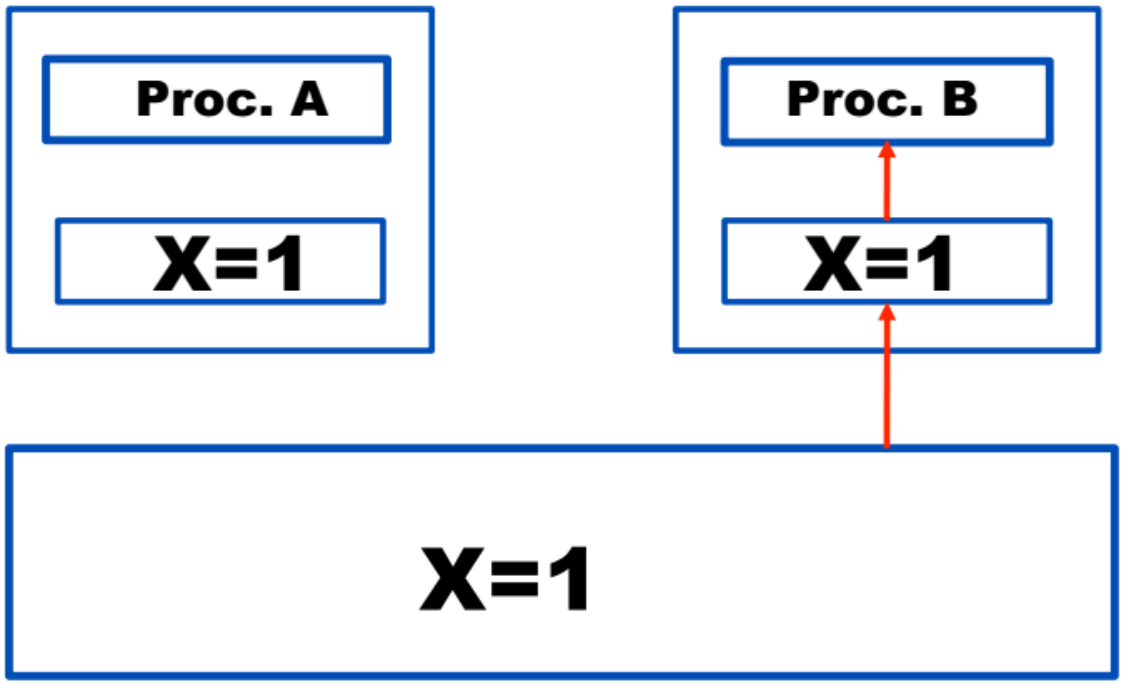
- Invalidate other copies on write



- Processor B reads X (cache miss)

Write Invalidate Cache Coherence

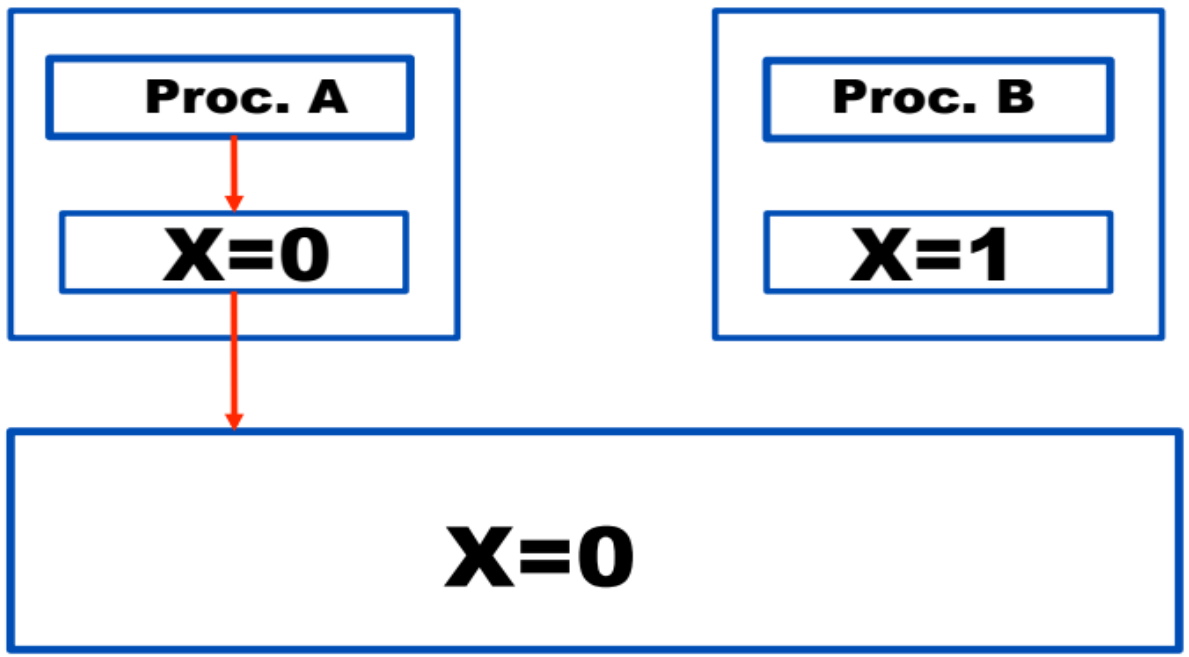
- Invalidate other copies on write:



- Processor B reads X (cache is updated)

Write Invalidate Cache Coherenc

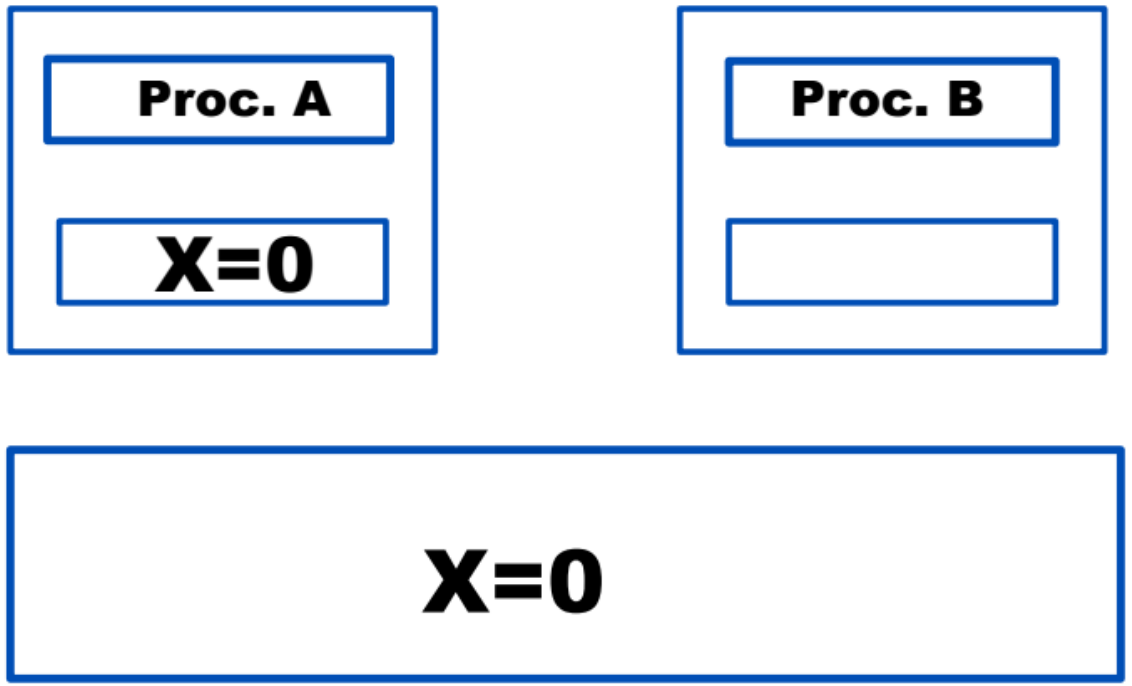
- Invalidate other copies on write:



- Processor A writes 0 into X (assume write-though cache)

Write Invalidate Cache Coherence

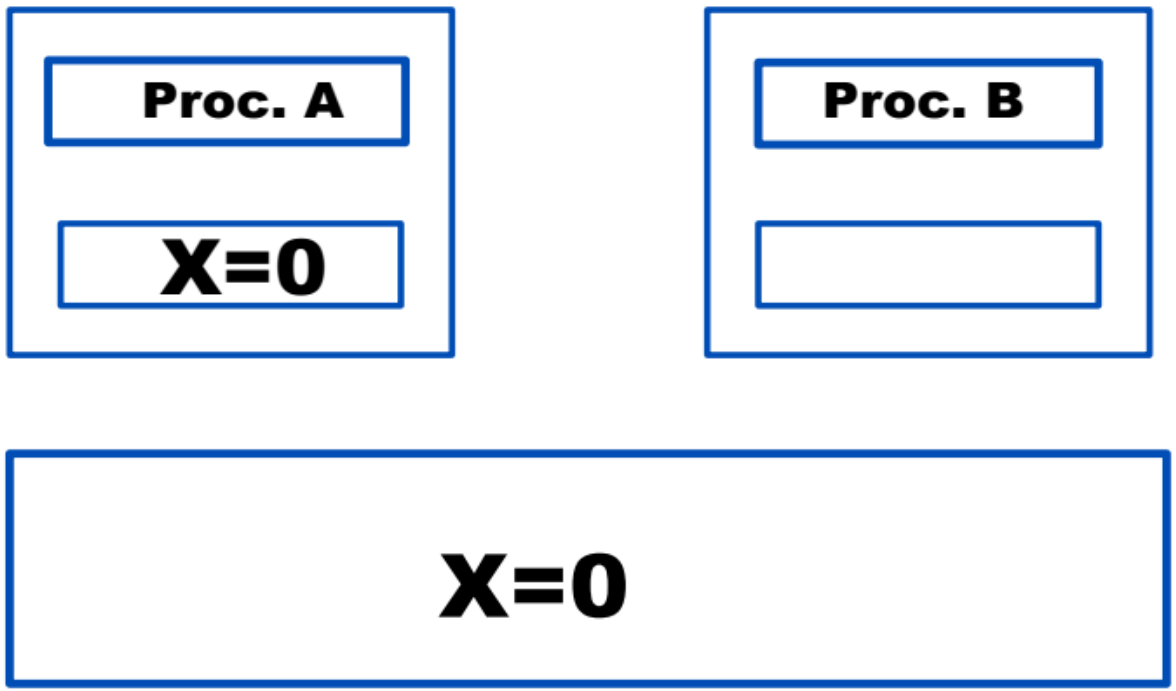
- Invalidate other copies on write:



- Copy of X in B's cache was invalidated.

Write Invalidate Cache Coherence

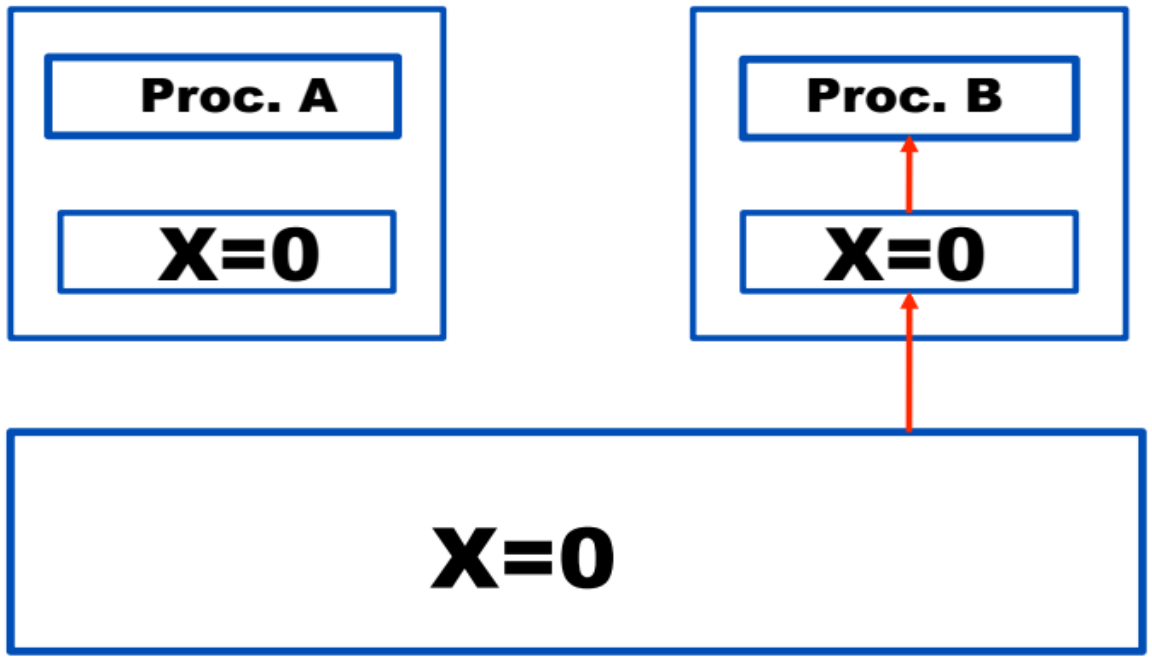
- Invalidate other copies on write



- Processor B reads X (cache miss)

Write Invalidate Cache Coherence

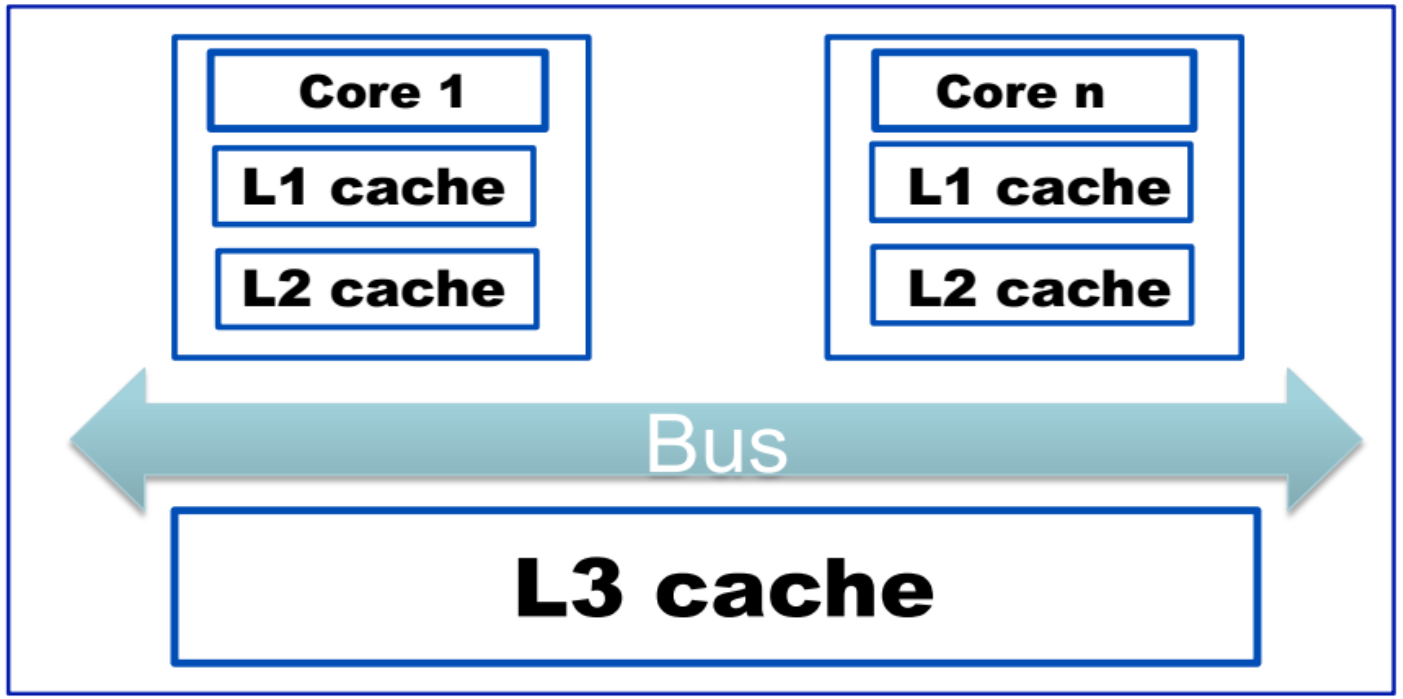
- Invalidate other copies on write:



- Processor B reads X

Basic Snooping Cache Implementation

- To invalidate: processor acquires the bus and broadcast access to be invalidated.



- All processors continuously snoop on the bus watching addresses. They invalidate their caches if they have the address.

Snooping Cache Protocol

- States of a block:
  - Invalid
  - Shared: potentially shared with other caches
  - Modified: updated in the private cache. Implies that the block is exclusive.

- Requests to the cache can come from a core or from the bus.

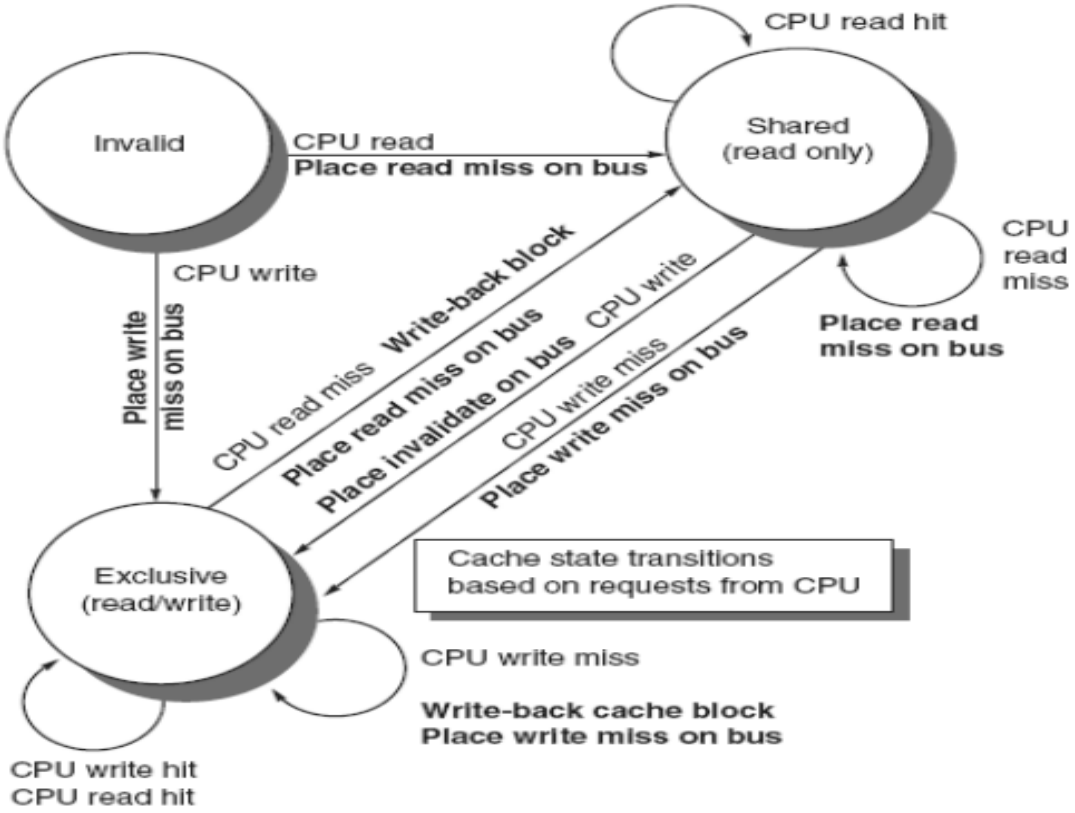
Snooping Coherence Protocols

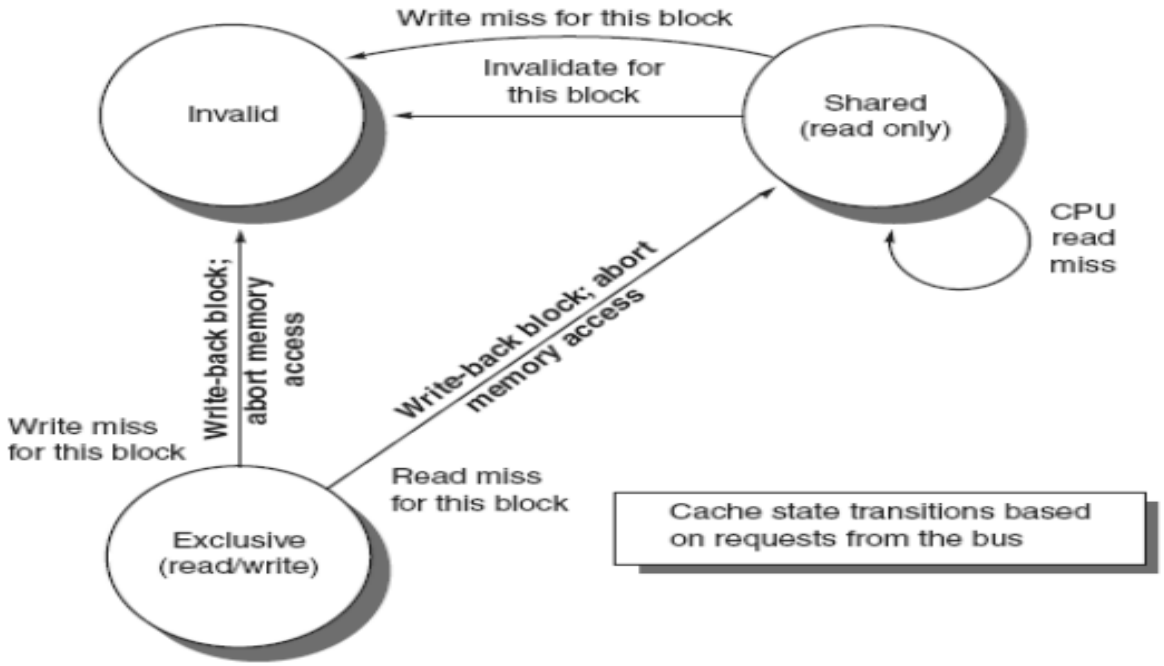
Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Snooping Coherence Protocols

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Write-Invalidate Cache Coherence Protocol For a Write-Back Cache





Circles: state of a block in the cache; Bold on transitions: bus actions generated as part of the state transition. Actions in parentheses allowed by local processor without state change.