Multiprocessor System

Presented By

Dr. Banchhanidhi Dash

School of Computer Engineering
KIIT University

Multiprocessor

- A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM. The main objective of using a multiprocessor is to boost the system's execution speed.

Multiprocessor system

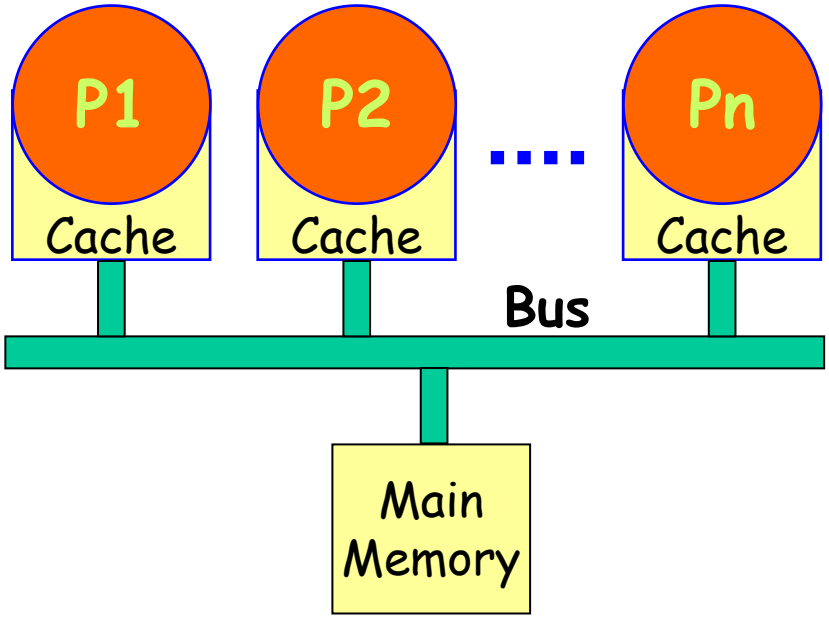
I. Centralized Shared-memory architecture (Tightly coupled multiprocessor)

II. Distributed Shared memory architecture (Loosely coupled multiprocessor)

Centralized Shared-memory architecture (Tightly coupled multiprocessor)

- ✓ **shared memory** is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors.
- ✓ Using memory for communication inside a single program, e.g. among its multiple threads, is also referred to as shared memory.
- ✓ In computer hardware, shared memory refers to a (typically large) block of random access memory (RAM) that can be accessed by several different central processing units (CPUs) in a multiprocessor computer system.
- ✓ Shared memory systems may use:
  - ✓ uniform memory access (UMA): all the processors share the physical memory uniformly;
  - ✓ non-uniform memory access (NUMA): memory access time depends on the memory location relative to a processor;

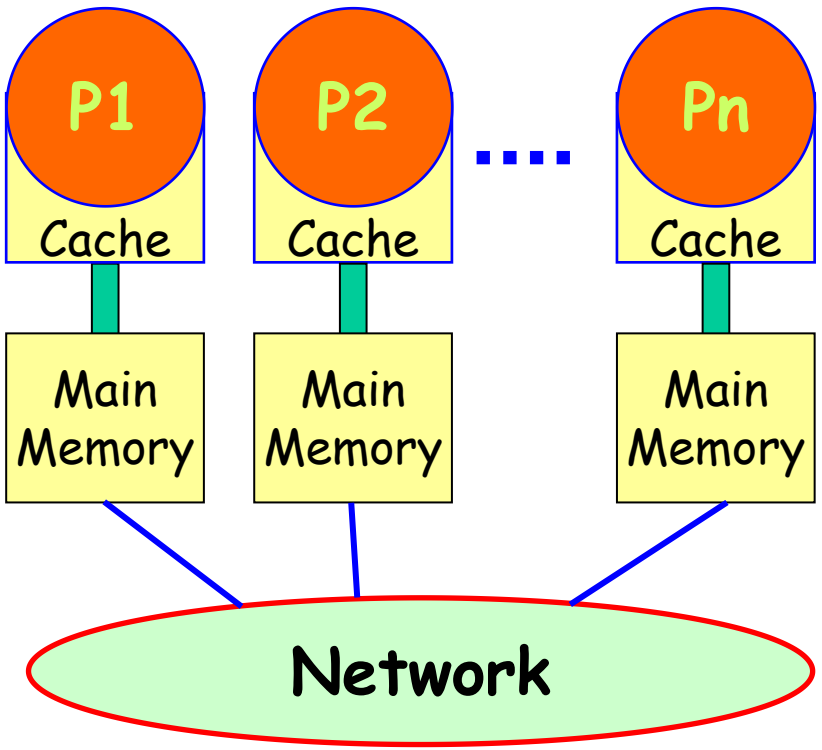
UMA vs. NUMA Computers



Latency = 100s of ns

(a) UMA Model

Latency = several milliseconds to seconds



(b) NUMA Model

The **issue with shared memory systems** is that many CPUs need fast access to memory and will likely cache memory, which has two complications:

lack of data coherence: whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data.

Distributed Shared memory architecture (Loosely coupled multiprocessor)

Distributed memory refers to a multiprocessor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data are required, the computational task must communicate with one or more remote processors.

Distributed Memory Computers

- Distributed memory computers use:
  - Message Passing Model
- Explicit message send and receive instructions have to be written by the programmer.
  - Send: specifies local buffer + receiving process (id) on remote computer (address).
  - Receive: specifies sending process on remote computer + local buffer to place data.

DSM

- Physically separate memories are accessed as one logical address space.
- Processors running on a multi-computer system share their memory.
  - Implemented by operating system.
- DSM multiprocessors are NUMA:
  - Access time depends on the exact location of the data.

Distributed Shared-Memory Architecture (DSM)

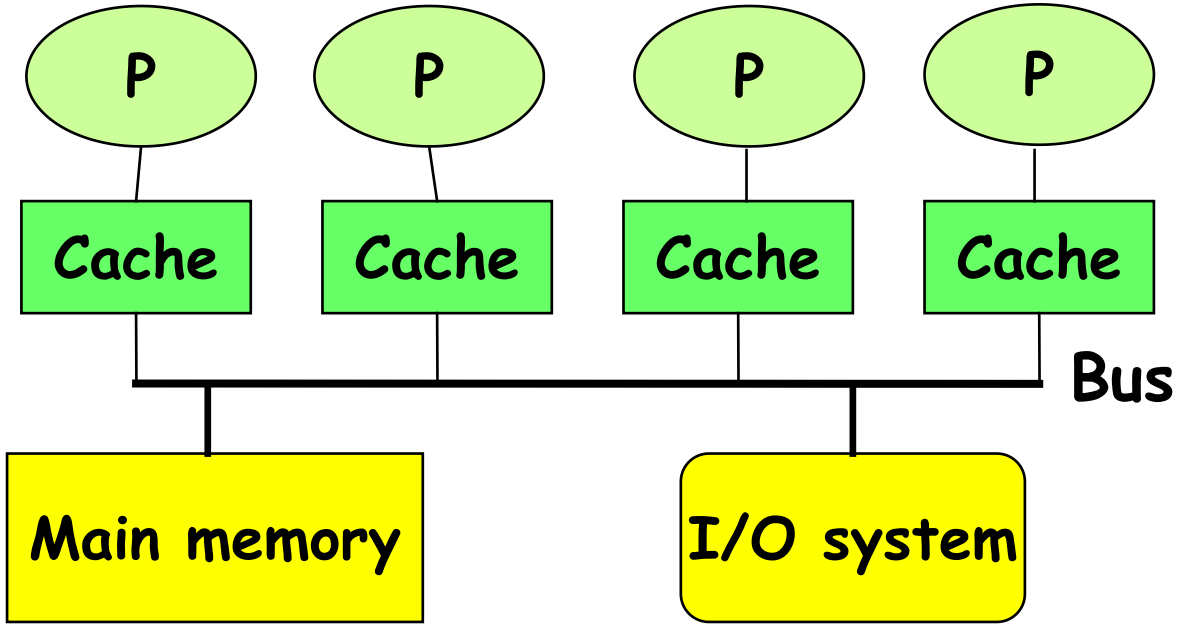
- Underlying mechanism is message passing:
  - Shared memory convenience provided to the programmer by the operating system.
  - Basically, an operating system facility takes care of message passing implicitly.
- Advantage of DSM:
  - Ease of programming

Disadvantage of DSM

- High communication cost:
  - A program not specifically optimized for DSM by the programmer shall perform extremely poorly.
  - Data (variables) accessed by specific program segments have to be collocated.
  - Useful only for process-level parallelism.

Symmetric Multiprocessors (SMPs)

- SMPs are a popular shared memory multiprocessor architecture:
  - Processors share Memory and I/O
  - Bus based: access time for all memory locations is equal --- "Symmetric MP"



An Important Problem with Shared-Memory: Coherence

- When shared data are cached:
  - These are replicated in multiple caches.
  - The data in the caches of different processors may become inconsistent.
- How to enforce cache coherency?
  - How does a processor know changes in the caches of other processors?

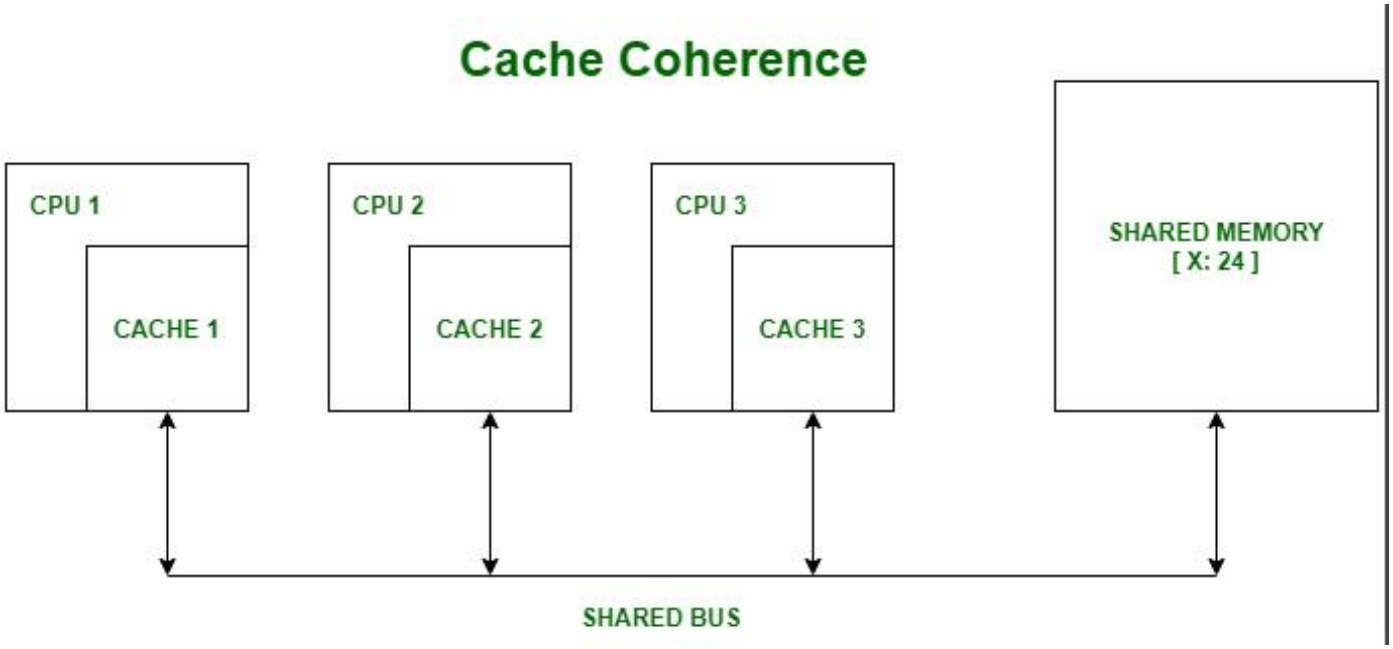
Cache Coherence:

- It is the uniformity of shared resource data that ends up stored in multiple local caches.

Cache Coherence Problem:

- It is the challenge of keeping multiple local caches synchronized when one of the processors updates its local copy of data which is shared among multiple caches.

cache coherence problem example



Suppose there are three processors, each having cache.
Suppose the following scenario:-
Processor 1 read X : obtains 24 from the memory and caches it.
Processor 2 read X : obtains 24 from memory and caches it.
Again, processor 1 writes as X : 64, Its locally cached copy is updated.
Now, processor 3 reads X, what value should it get?
Memory and processor 2 thinks it is 24 and processor 1 thinks it is 64.

As multiple processors operate in parallel, and independently multiple caches may possess different copies of the same memory block, this creates a **cache coherence problem.**

Two Important Cache Coherency Protocols

- Snooping protocol:
- Directory-based protocol:

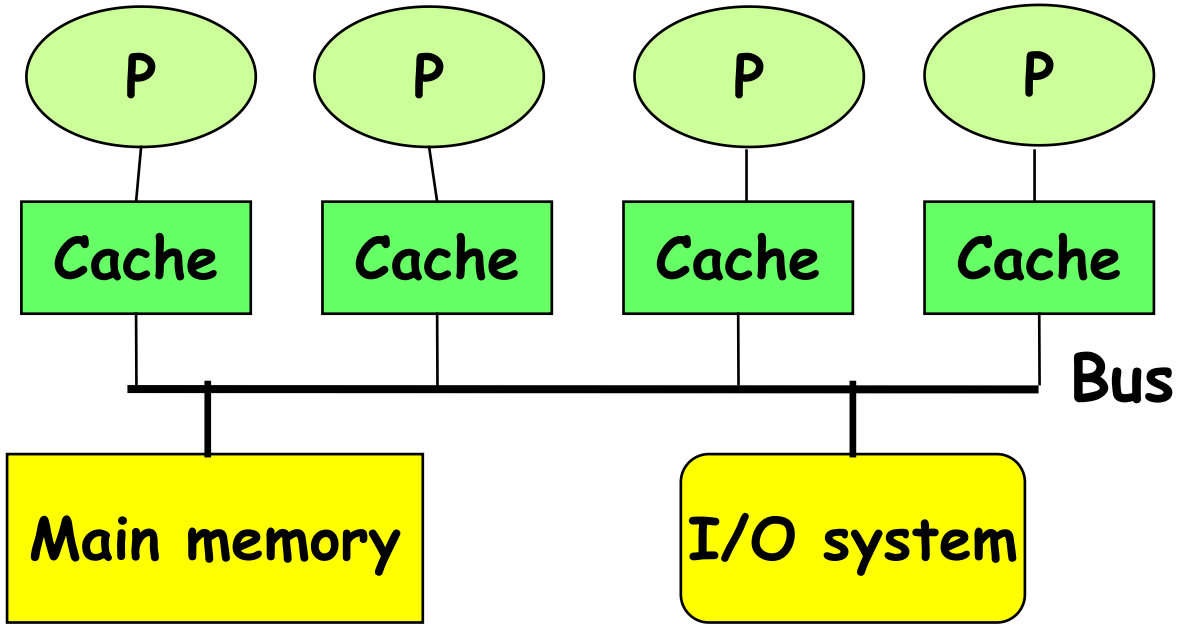
Snooping Protocol

- Snooping protocol:
  - Each cache "snoops" the bus to find out which data is being used by whom.
- As soon as a request for any data block by a processor is put out on the bus:
  - Other processors "snoop" to check if they have a copy and respond accordingly.
- Works well with bus interconnection:
  - All transmissions on a bus are essentially broadcast:
    - Snooping is therefore effortless.
  - Dominates almost all small scale machines.

Categories of Snoopy Protocols

- Essentially two types:
  - Write Invalidate Protocol
  - Write Broadcast Protocol/write update protocol
- Write invalidate protocol:
  - When one processor writes to its cache, all other processors having a copy of that data block invalidate that block.
- Write broadcast:
  - When one processor writes to its cache, all other processors having a copy of that data block update that block with the recent written value.

Write Invalidate Vs. Write Update Protocols



Write Invalidate Protocol

- Handling a write to shared data:
  - An invalidate command is sent on bus --- all caches snoop and invalidate any copies they have.
- Handling a read Miss:
  - Write-through: memory is always up-to-date.
  - Write-back: snooping finds most recent copy.

Write Invalidate in Write Through Caches

- Simple implementation.
- Writes:
  - Write to shared data: broadcast on bus, processors snoop, and update any copies.
  - Read miss: memory is always up-to-date.
- Concurrent writes:
  - Write serialization automatically achieved since bus serializes requests.
  - Bus provides the basic arbitration support.

Write Invalidate versus Broadcast cont…

- Invalidate exploits spatial locality:
  - Only one bus transaction for any number of writes to the same block.
  - Obviously, more efficient.
- Broadcast has lower latency for writes and reads:
  - As compared to invalidate.

Implementation of the Snooping Protocol

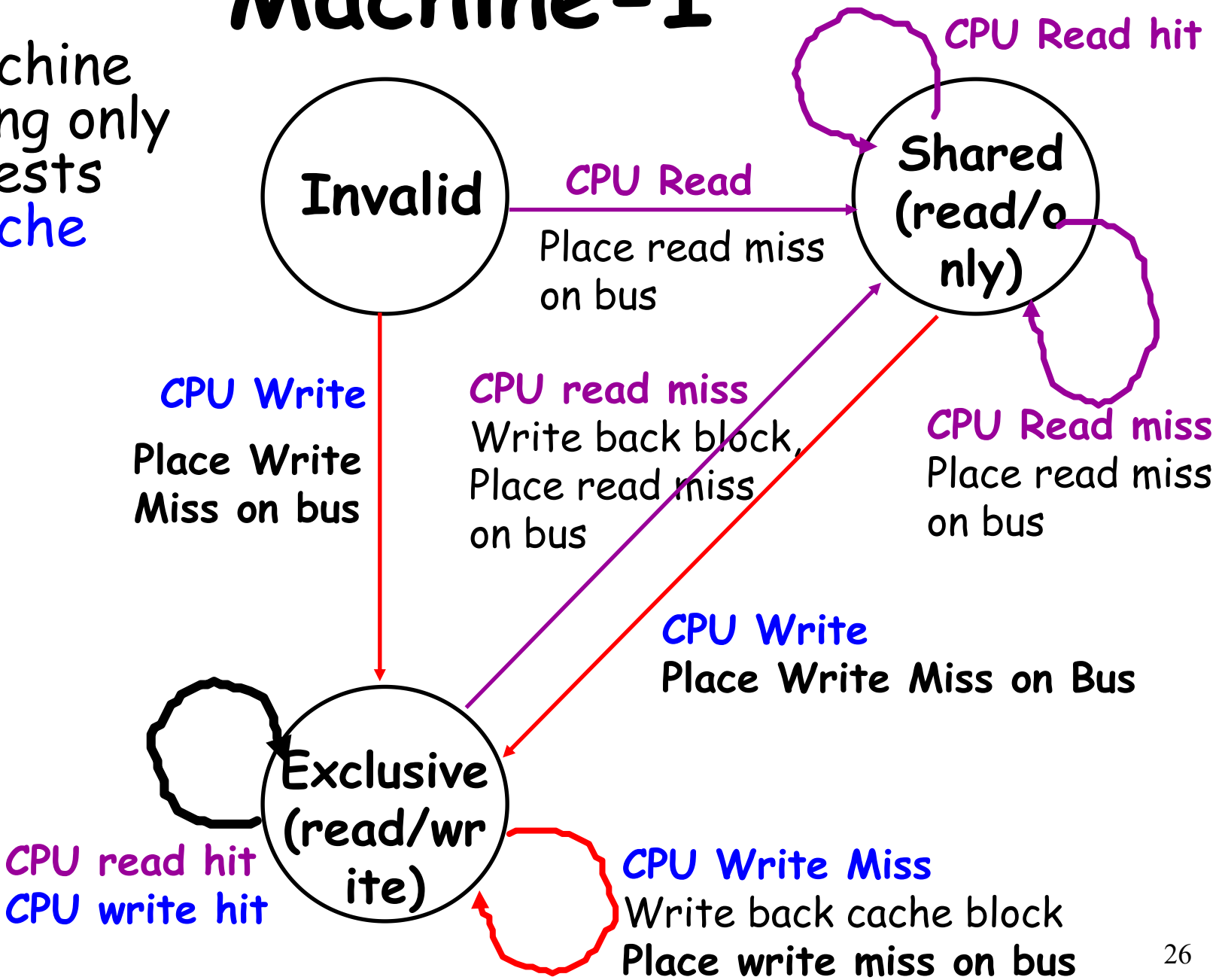
- 3 state in snoopy protocol
  - Shared: Clean in all caches and up-to-date in memory, block can be read.
  - Exclusive: cache has the only copy, it is writeable, and dirty.
  - Invalid: Data present in the block obsolete, cannot be used.

Implementation of the Snooping Protocol

- A cache controller is responsible for interfacing to the lower level of memory when the access misses in order to fill the cache.
- A cache controller at every processor would implement the protocol:
  - Has to perform specific actions:
    - When the local processor requests certain things.
    - Also, certain actions are required when certain address appears on the bus.
  - Exact actions of the cache controller depends on the state of the cache block.
  - Two FSMs can show the different types of actions to be performed by a controller.

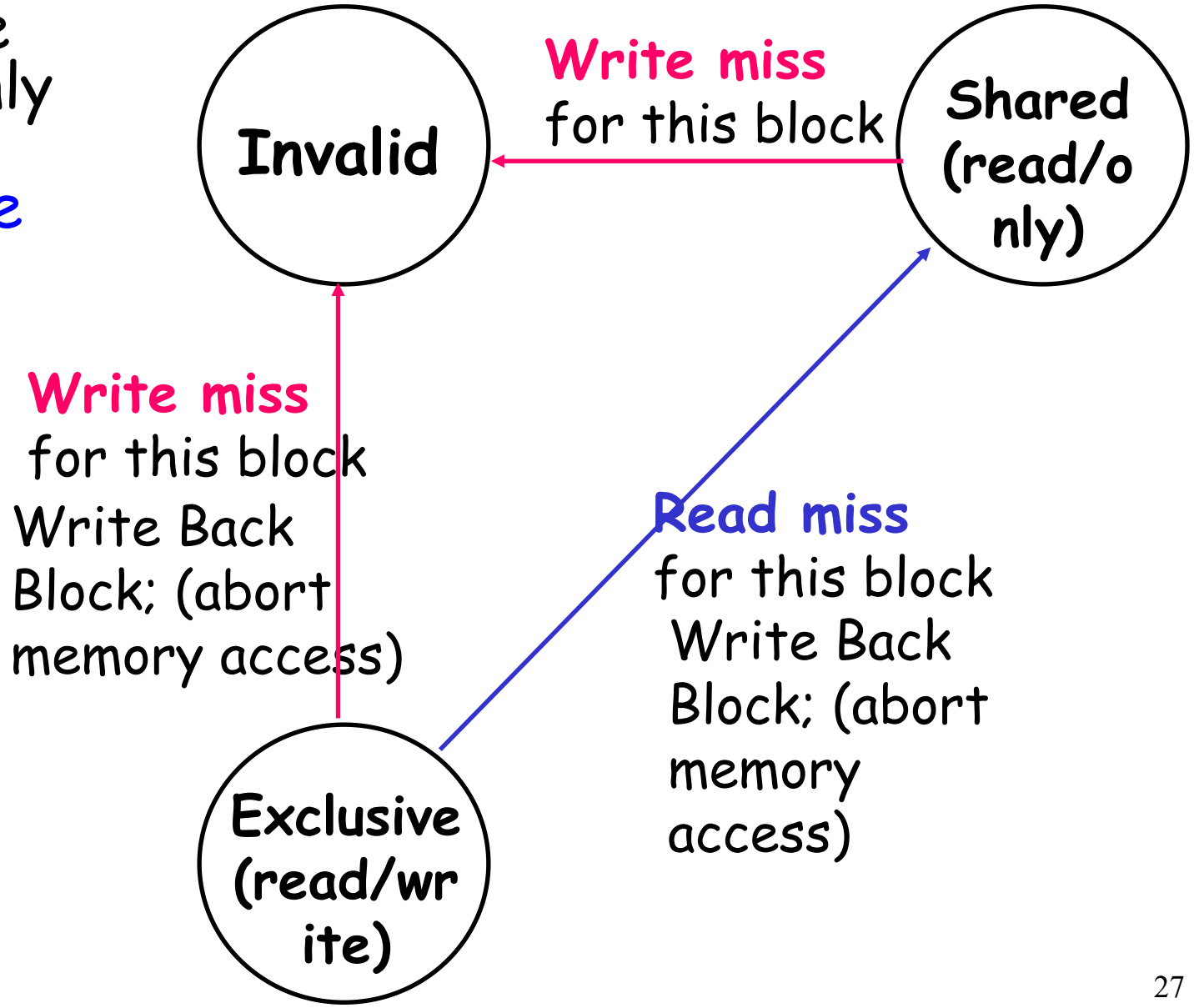
Snoopy-Cache State Machine-I

- State machine considering only CPU requests a each cache block.



Snoopy-Cache State Machine-II

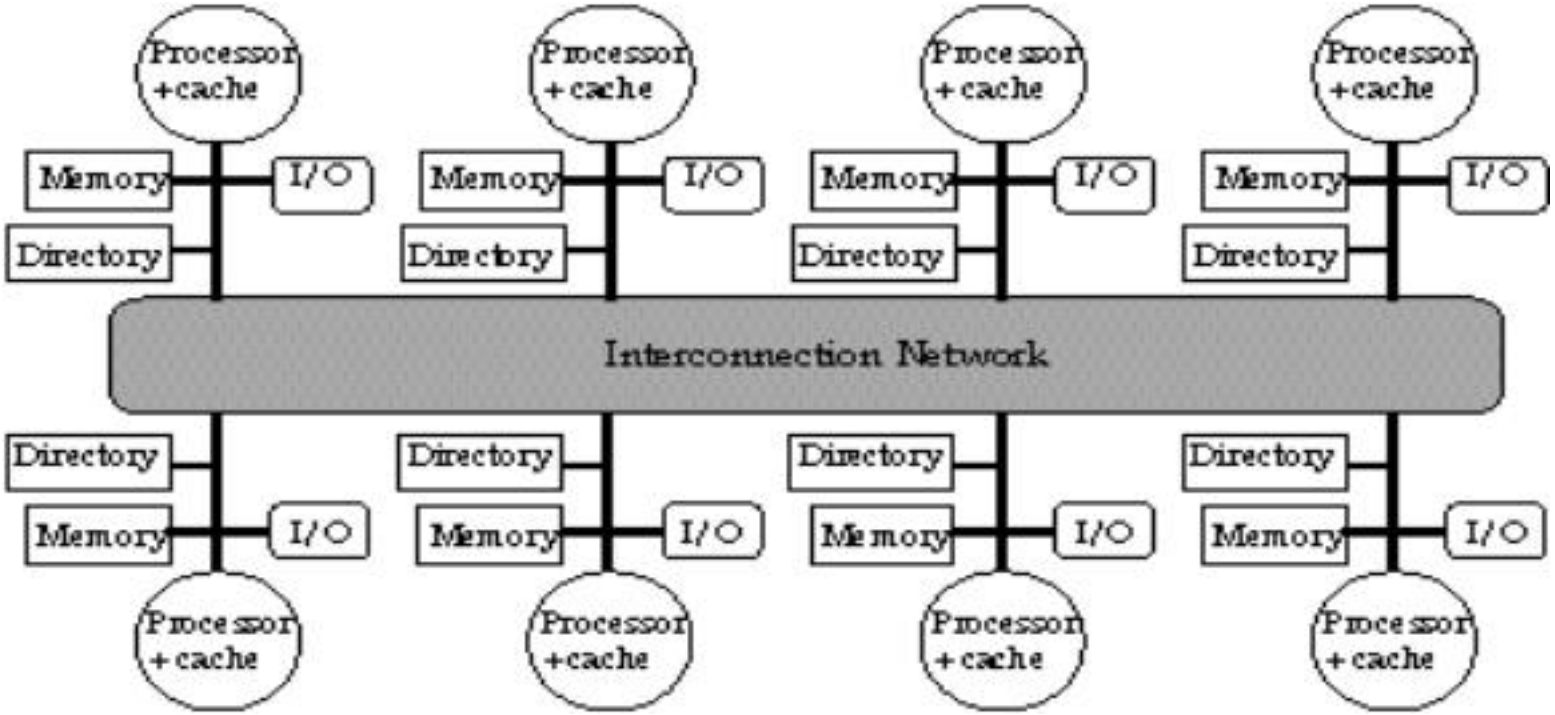
- State machine considering only bus requests for each cache block.



Directory-based Solution

- In NUMA computers:
  - Messages have long latency.
  - Also, broadcast is inefficient --- all messages have explicit responses.
- Main memory controller to keep track of:
  - Which processors are having cached copies of which memory locations.
- On a write,
  - Only need to inform users, not everyone
- On a dirty read,
  - Forward to owner

Distributed Directory multiprocessors



- directory per cache that tracks state of every block in every cache

Directory Protocol

- Three states as in Snoopy Protocol
  - Shared: 1 or more processors have data, memory is up-to-date.
  - Uncached: No processor has the block.
  - Exclusive: 1 processor (owner) has the block.
- In addition to cache state,
  - In addition to cache state, a directory must track which processors have data when in the shared state. This is required for sending invalidation and intervention requests to the individual processor caches which have the cache block in shared state.
  -

Directory Behavior

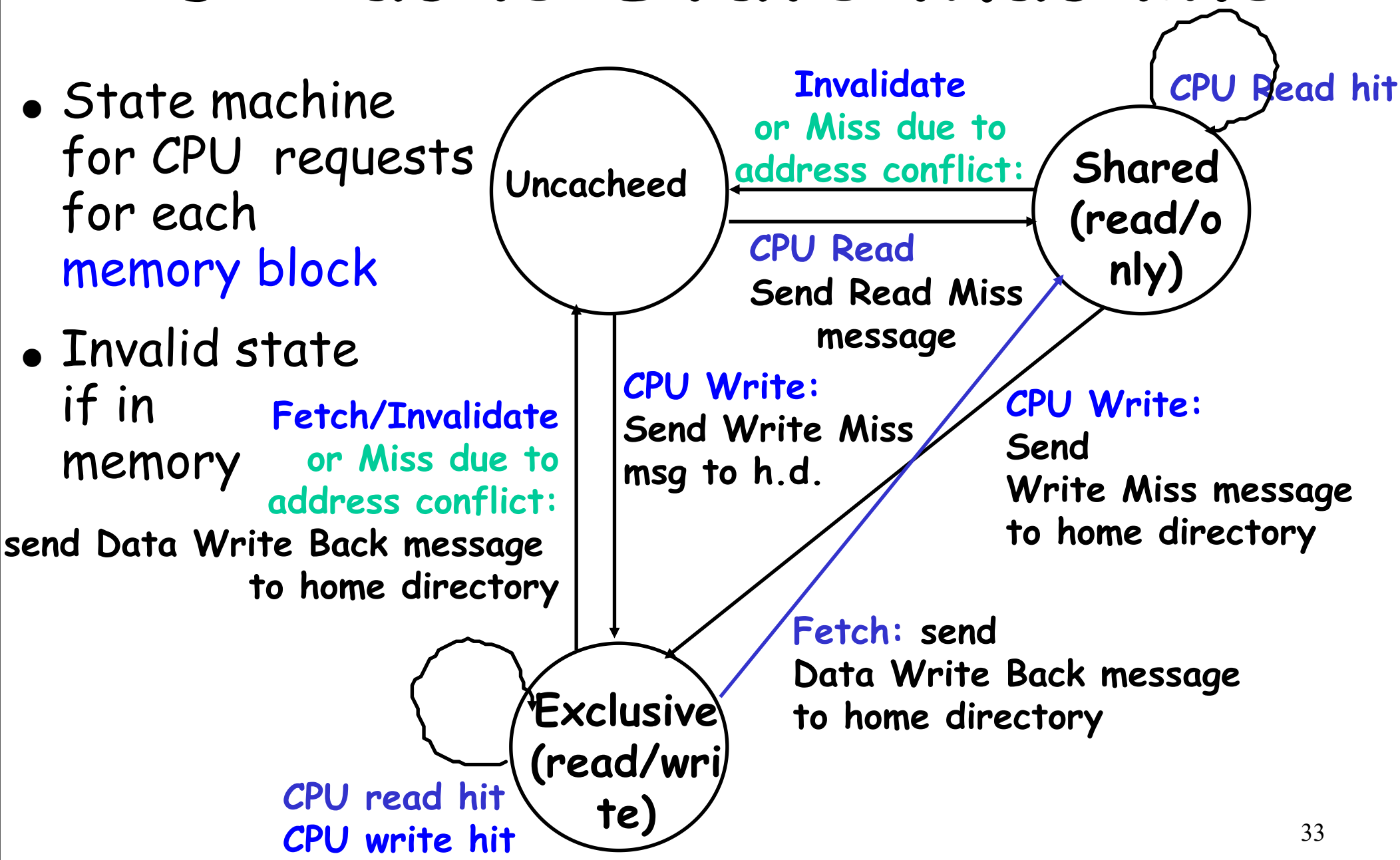
- On a read:
  - Unused:
    - give (exclusive) copy to requester
    - record owner
  - Exclusive or shared:
    - send share message to current exclusive owner
    - record owner
    - return value
  - Exclusive dirty:
    - forward read request to exclusive owner.

Directory Behavior

- On Write
  - Send invalidate messages to all hosts caching values.
- On Write-Thru/Write-back
  - Update value.

CPU-Cache State Machine

- State machine for CPU requests for each memory block
- Invalid state if in memory



State Transition Diagram for the Directory

- Tracks all copies of memory block.
- Same states as the transition diagram for an individual cache.
- Memory controller actions:
  - Update of directory state
  - Send msgs to statisfy requests.
  - Also indicates an action that updates the sharing set, Sharers, as well as sending a message.

Directory State Machine

- State machine for Directory requests for each memory block
- Uncached state if in memory