

# CHAPTER 7: DISTRIBUTED SHARED MEMORY

DSM simulates a logical shared memory address space over a set of physically distributed local memory systems.

## Why DSM?

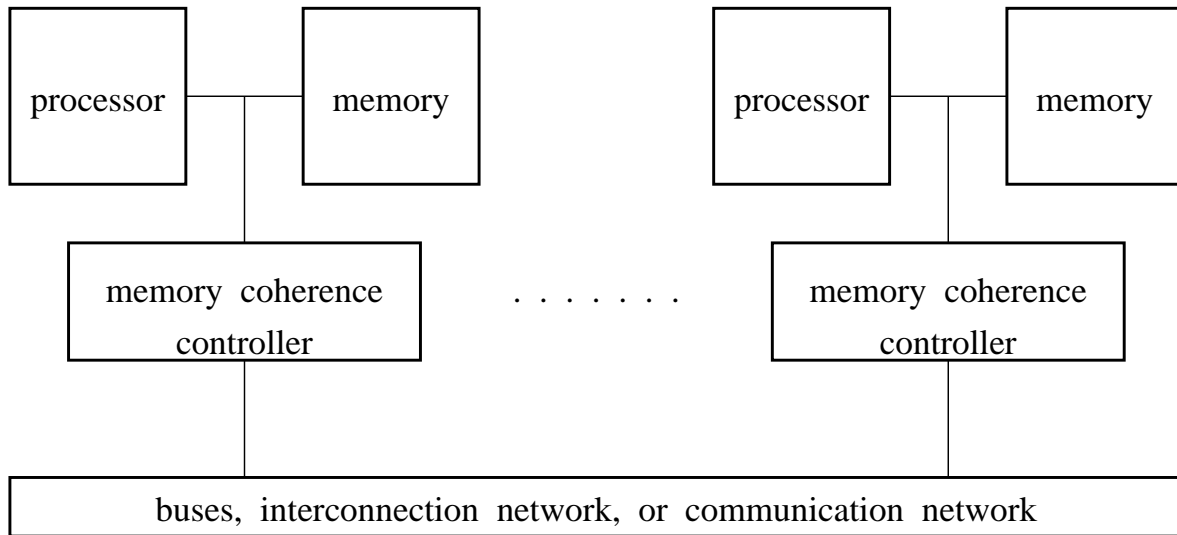
- direct information sharing programming paradigm (transparency)
- multilevel memory access (locality)
- wealth of existing programs (portability)
- large physical memory
- scalable multiprocessor system

## Chapter outline

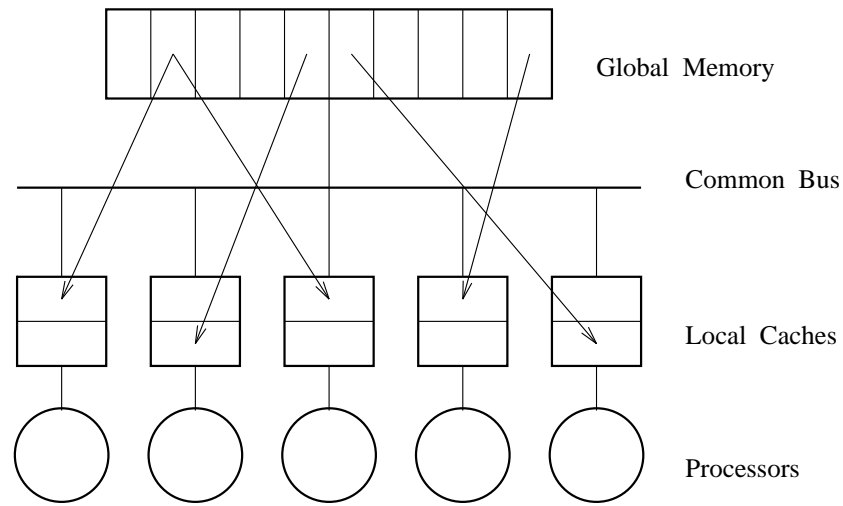
- NUMA architectures: similarity between multiprocessor cache and DSM systems
- Memory consistency models: why is memory consistency a more critical problem in multiprocessor and DSM systems? how is memory consistency defined?
- Cache coherency protocols: implementation of consistency models
- DSM Implementation: applying the consistency models and coherency protocols to a DSM system

# Nonuniform Memory Access (NUMA) architectures

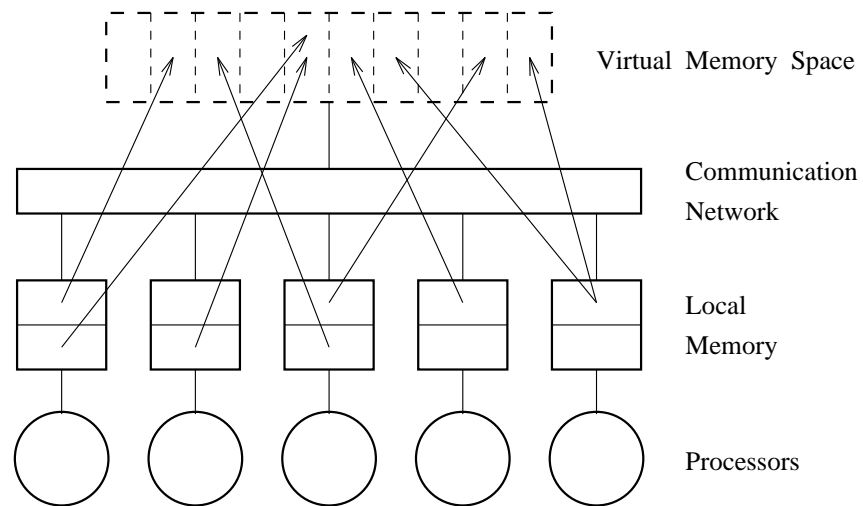
## Generic NUMA architecture



## Multiprocessor Cache and DSM architectures



(a) Multiprocessor cache architecture



(b) Distributed shared memory architecture

## Common issues

Data consistency and coherency due to data placement, migration and replication

- Data Sharing Granularity
- Cache Miss Granularity
- Tradeoffs:
  - Transfer time
  - Administrative overhead
  - Hit rate
  - Replacement rate
  - False Sharing

What to do on cache miss?

- Locating block - owner/directory
- Block Migration - block bouncing
- Block Replication
- Push vs. Pull

## Memory consistency models

These models apply consistency constraints to all memory accesses  
 Accesses may require multiple messages and take significant time

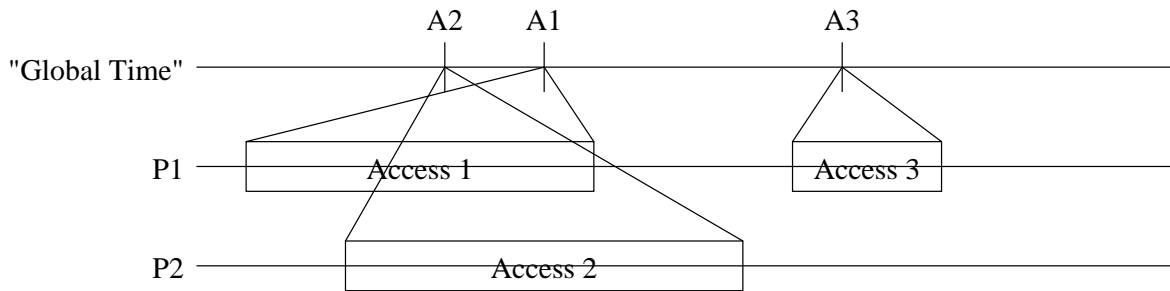
### Atomic consistency

Respect real-time order

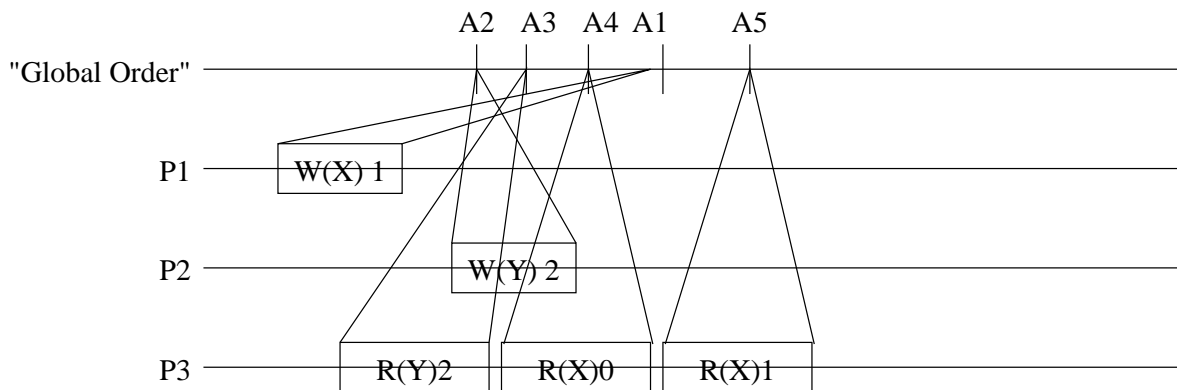
### Sequential consistency

All processors see same order and  
 order respects all internal orders (not nec. real time)

$P_1$  :  $W(X)1$   
 $P_2$  :  $W(Y)2$   
 $P_3$  :  $R(Y)2$   $R(X)0$   $R(X)1$



Atomic Consistency – global total order respecting access intervals

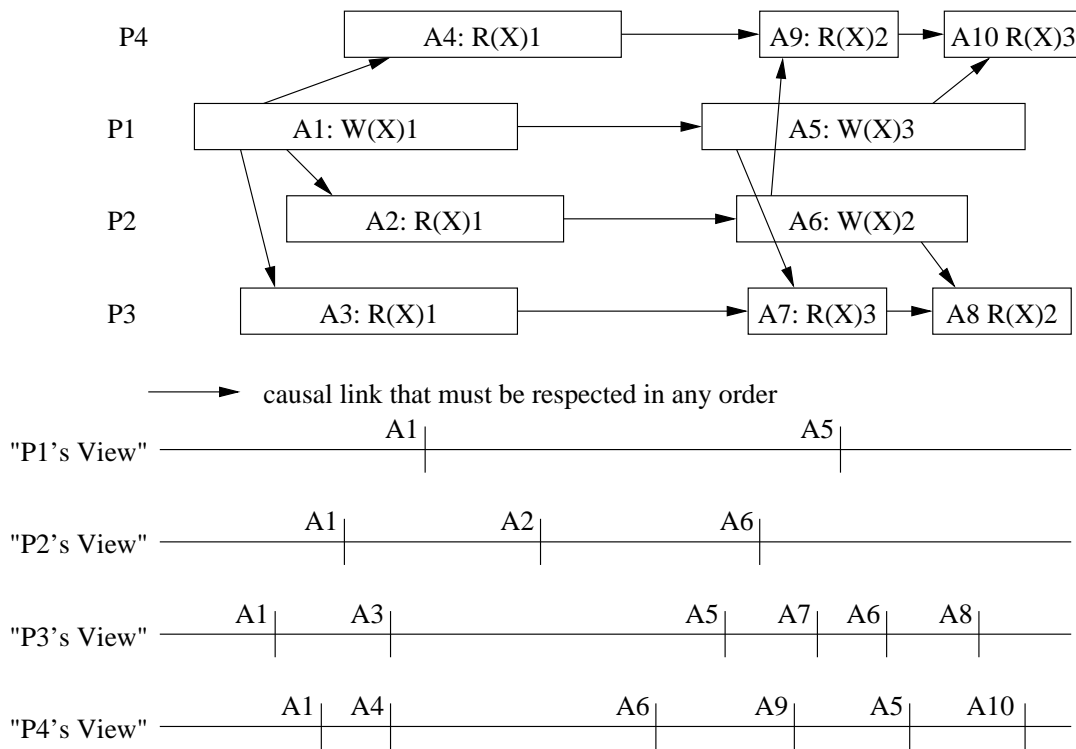


Sequential Consistency – global total order (not nec. respecting access intervals)

## Causal consistency

Processors may see different order  
all orders respect causal order

$P_1$  :  $W(X)1$                        $W(X)3$   
 $P_2$  :  $R(X)1$     $W(X)2$   
 $P_3$  :  $R(X)1$                        $R(X)3$     $R(X)2$   
 $P_4$  :  $R(X)1$                        $R(X)2$     $R(X)3$



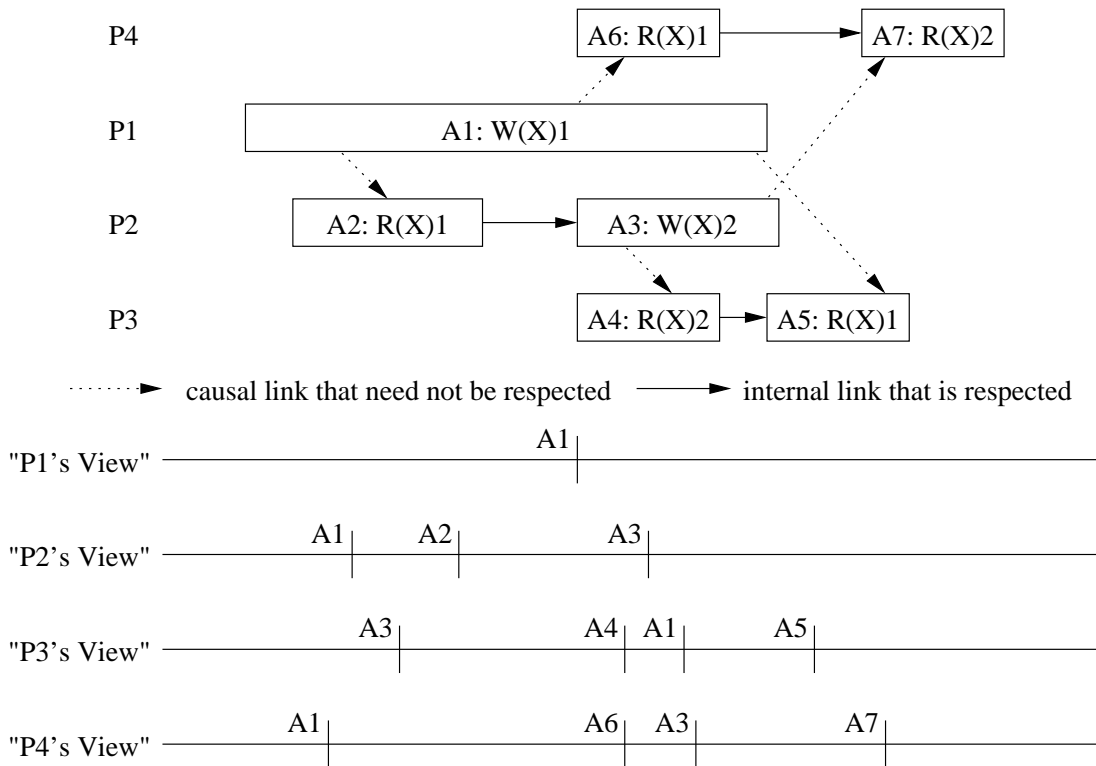
Causal Consistency – no global total order; causal partial order only  
Each processor's order respects internal order and Write-Read causality

## Processor consistency

Writes from same processor are in order

Writes from different processors not constrained

$P_1$  :  $W(X)1$   
 $P_2$  :  $R(X)1$   $W(X)2$   
 $P_3$  :  $R(X)1$   $R(X)2$   
 $P_4$  :  $R(X)2$   $R(X)1$



Processor Consistency – no global total order; partial order on writes by same processor

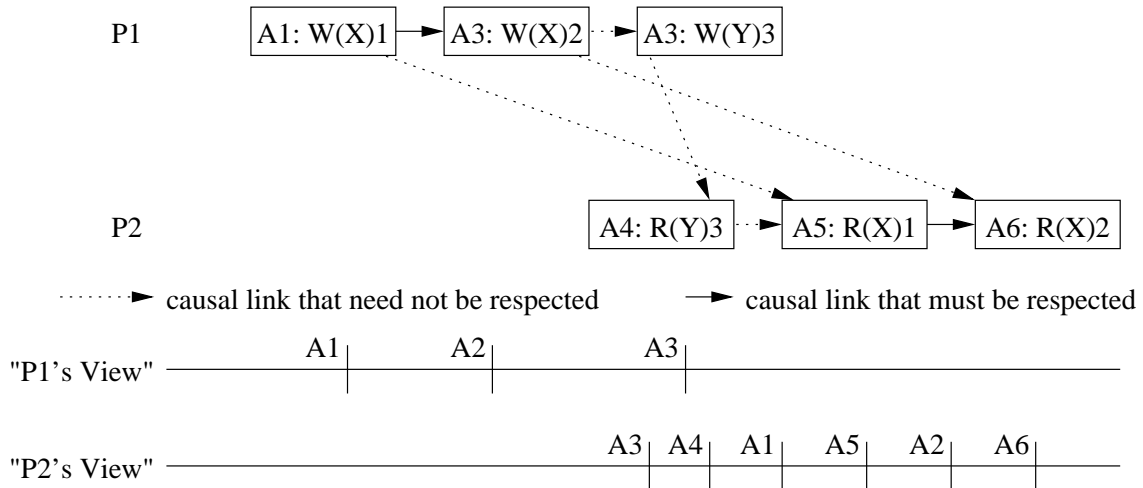
Each processor's order respects internal order and order of writes by same processor

## Slow memory consistency

Writes from same processor to same location are in order

Writes from different processors or locations not constrained

$P_1$  :  $W(X)1$   $W(X)2$   $W(Y)3$   
 $P_2$  :  $R(Y)3$   $R(X)1$   $R(X)2$



Slow Memory Consistency – no global total order, no constraints across memory locations

Each processor's order respects its internal order and order of writes to same memory by same processor

# Synchronization Access Consistency Models

## Weak consistency

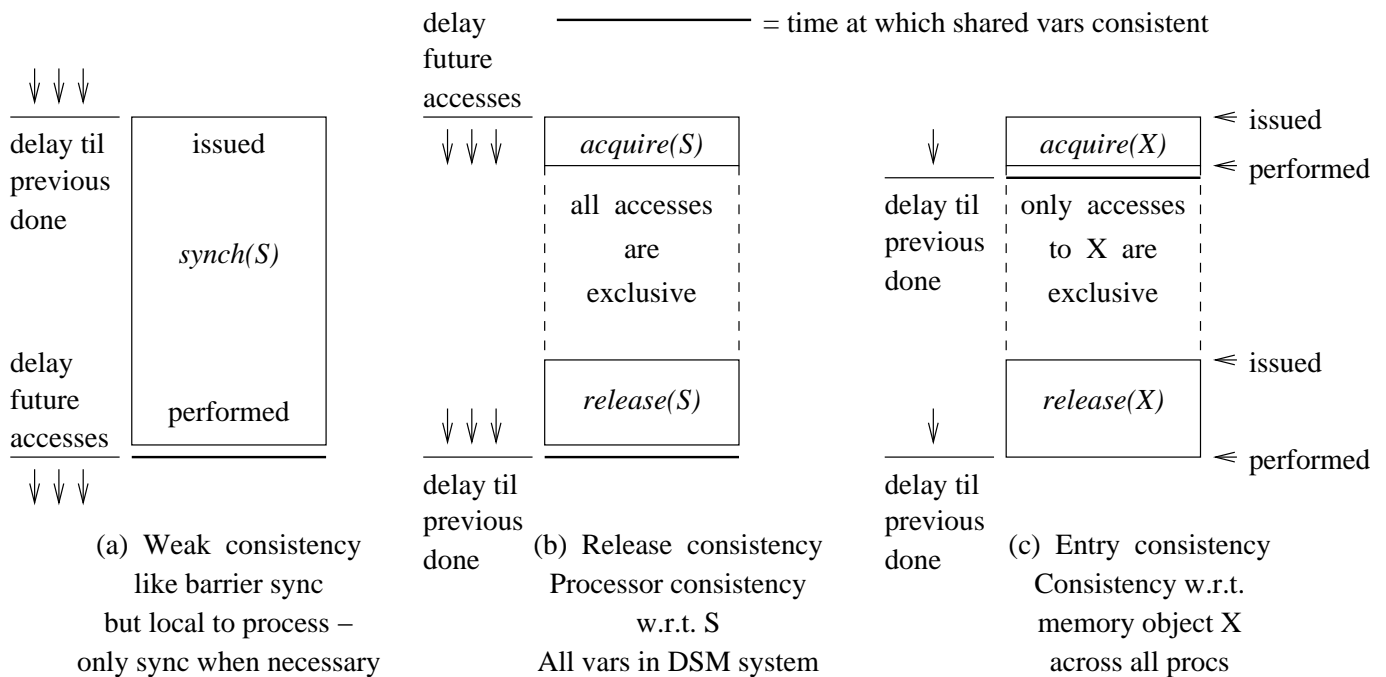
- Accesses to *synchronization variables* are sequentially consistent
- No access to a synchronization variable is issued by a processor before all previous *read/write* data accesses have been performed
- No *read/write* data access is issued by a processor before a previous access to a synchronization variable has been performed

## Release consistency

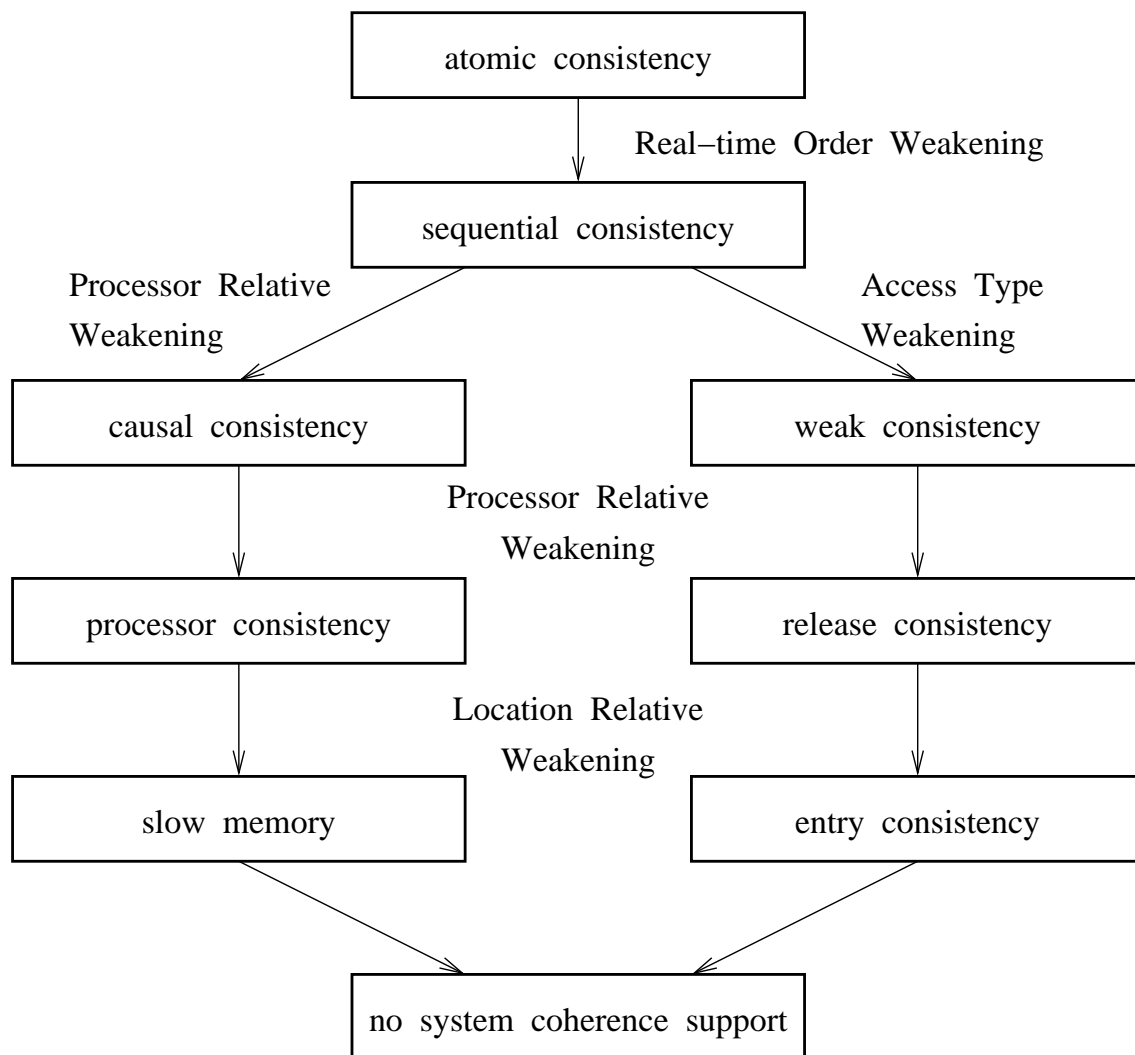
The synchronization access ( $synch(S)$ ) in the weak consistency model can be refined as a pair of  $acquire(S)$  and  $release(S)$  accesses. Shared variables in the critical section are made consistent when the release operation is performed.

## Entry consistency

$acquire$  and  $release$  are applied to general variables.

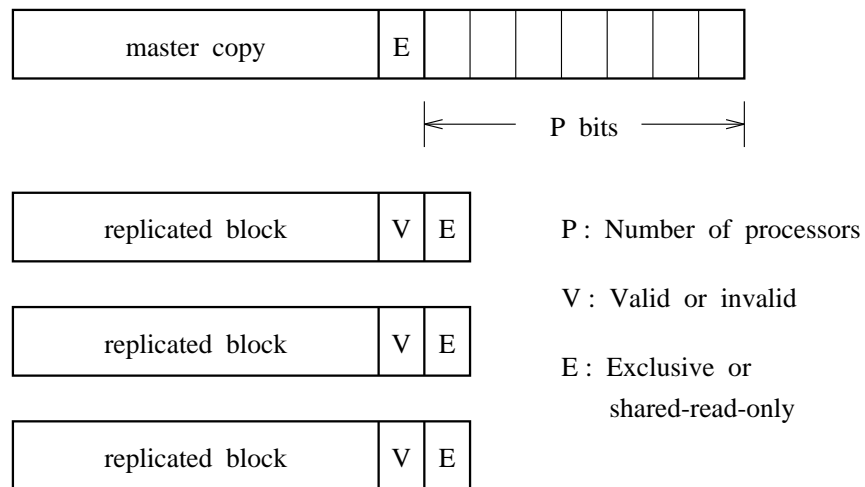


# Taxonomy



# Multiprocessor Cache Systems

## Cache directory



May also include *private* bit and/or *dirty* bit.

## Cache coherency protocols

### write-invalidate and write-update

#### Write-invalidate

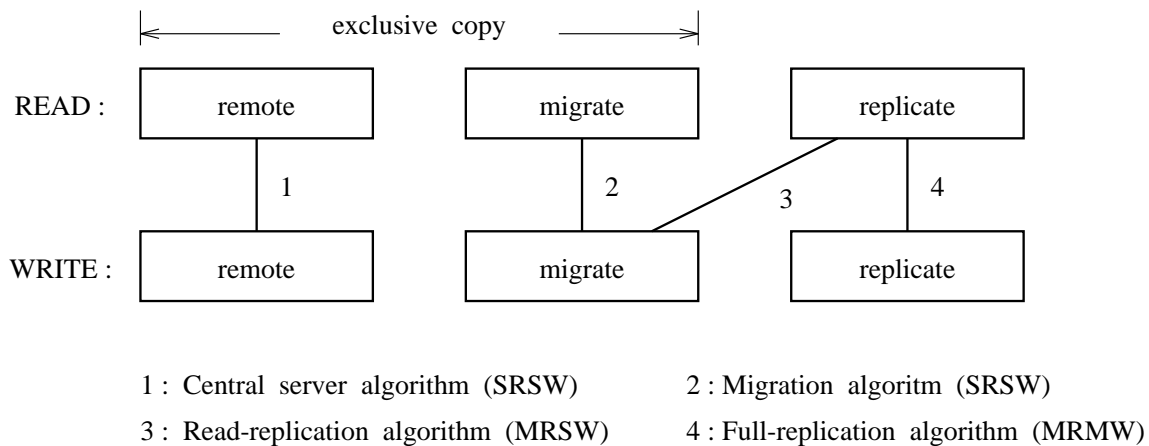
- Read hit
- Read miss: transfer block, set P-, V-, and E-bit.
- Write hit: invalidate cache copies, write and set E-bit
- Write miss

#### Hardware mechanisms

- Directory-based
- Snooping cache

## DSM implementation

### Memory management algorithms



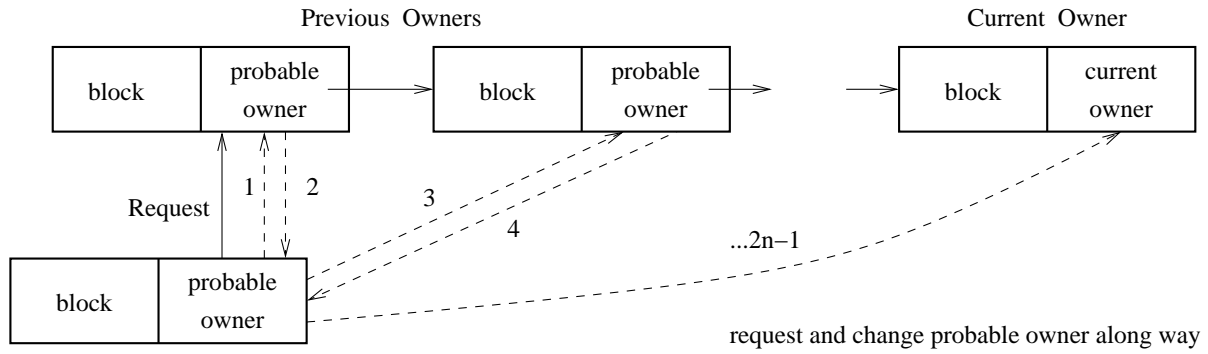
- Read-remote-write-remote: long network delay, trivial consistency
- Read-migrate-write-migrate: thrashing and false sharing
- Read-replicate-write-migrate: write-invalidate
- Read-replicate-write-replicate; full concurrency, atomic update

#### Considerations:

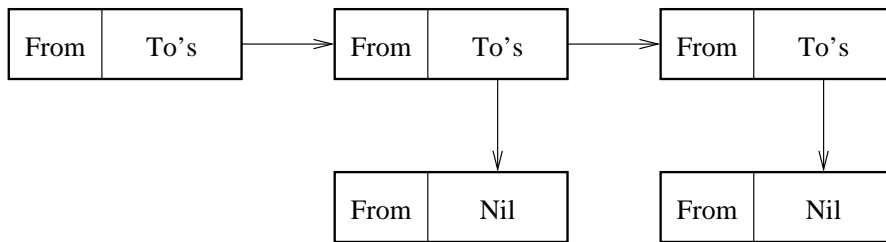
- Block granularity
- Block transfer communication overhead
- Read/write ratio
- Locality of reference
- Number of nodes and type of interaction

## Distributed implementation of Directory

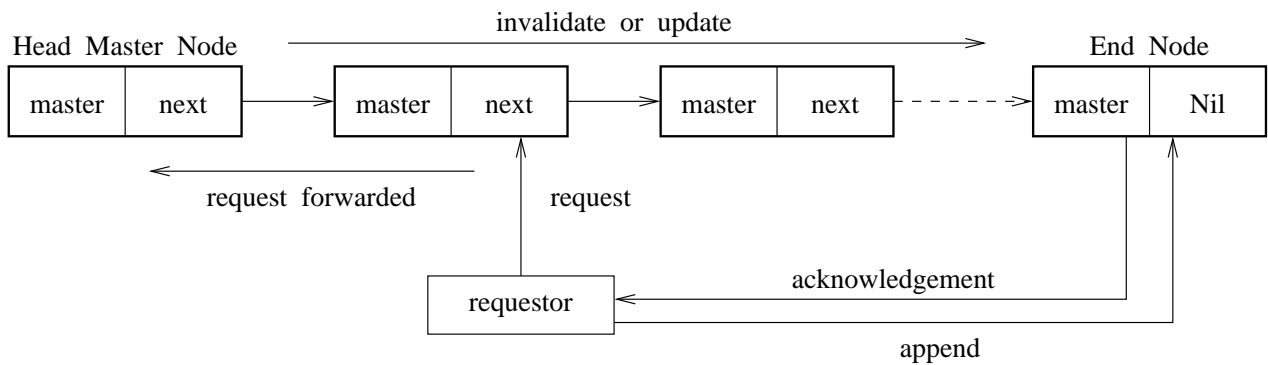
Locating Block Owner:



Maintaining Copy List:



(a) Spanning tree representation of copy set



(b) Linked list representation of copy set