Distributed Shared Memory

Distributed Shared Memory Overview

1. Must decide placement
   - Place memory (pages) close to relevant processes
2. Must decide migration
   - When to copy memory (pages) from remote to local
3. Must decide sharing rules
   - Ensure memory operations are properly ordered

DFS Review

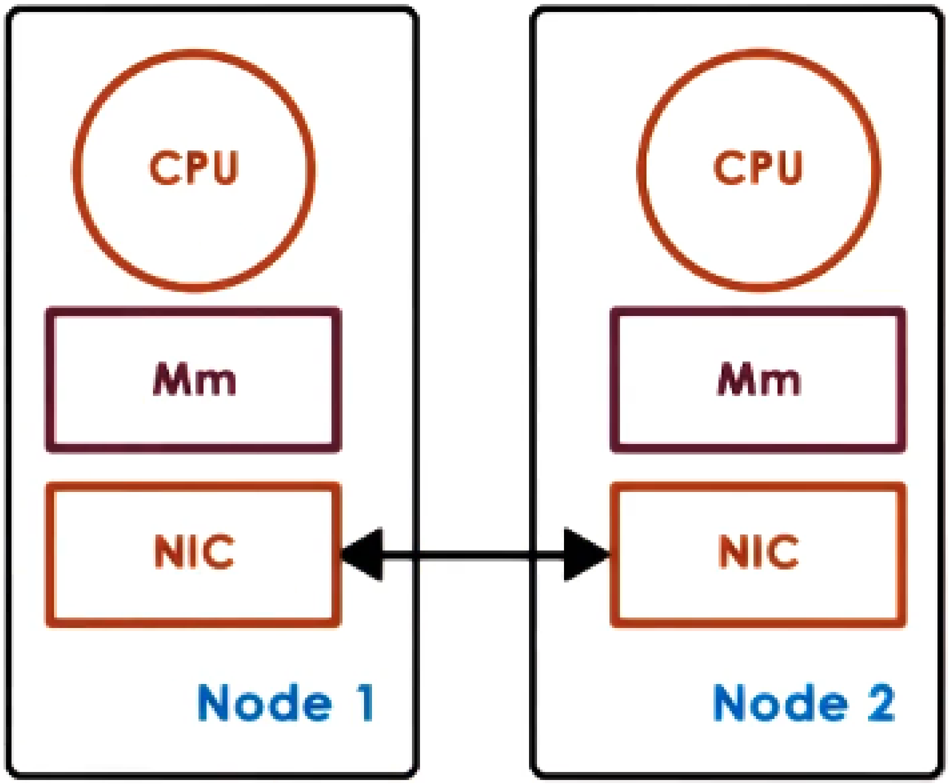
1. Clients
   - Send requests to file service
2. Caching
   - Improve performance (seen by clients) and scalability (supported by servers)
3. Servers
   - Own and manage state (files)
   - Provide service (file access)
4. How do we coordinate access of shared state among multiple servers?

Peer Distributed Applications

1. Each node…
   - "Owns" state (state is locally stored or generated)
   - Provides service
   - All nodes are "peers"
2. Examples: big data analytics, web searches, content sharing, or distributed shared memory (DSM)
3. In "peer-to-peer," overall management is done by all nodes

Distributed Shared Memory

1. Each node…
   - "Owns" state -> memory
   - Provides service
     - Memory reads/writes from any node
     - Consistency protocol
2. Permits scaling beyond single machine limits
   - More "shared" memory at lower cost
   - Slower overall memory access for remote memory
   - Commodity interconnect technologies offer low latency among nodes
     - RDMA: Remote Direct Memory Access



Overview of Distributed Shared Memory

Hardware vs Software DSM

1. Hardware supported (expensive!)
   - Relies on interconnect
   - OS manages larger physical memory
   - NICs translate remote memory accesses to messages
   - NICs involved in all aspects for memory management; support atomics...
2. Software supported
   - Everything done by software
   - OS, or language runtime
3. According to the paper "Distributed Shared Memory: Concepts and Systems", what is a common task that's implemented in software in hybrid (HW+SW) DSM implementations?
   - Prefetch pages - Easier to implement in software
   - Address translation - Easier to implement in hardware
   - Triggering invalidations - Easier to implement in hardware

DSM Design: Sharing Granularity

1. Cache line granularity? (used in SMP systems)
   - Overheads too high for DSM
2. Variable granularity
   - Overheads too high for small variables (integers)
3. Page granularity (OS-level)
4. Object granularity (language runtime)
5. Beware of false sharing

- Process 1 writes X, process 2 writes Y; both to separate locations
  - If X and Y are on the same page, coherence overhead is incurred
  - Try to put variables on separate pages (programmer or compiler)

DSM Design: Access Algorithm

1. Application access algorithm
   - Single reader/single writer (SRSW)
   - Multiple readers/single writer (MRSW)
   - Multiple readers/multiple writers (MRMW)
     - Writes must be correctly ordered to present a consistent view

DSM Design: Migration vs Replication

1. DSM performance metric == access latency
2. Migration: Copy state from one node to another as needed
   - Makes sense for SRSW
   - Requires data movement
   - Copying state for a single R/W is expensive (not amortized)
3. Replication: State is copied across multiple (potentially all) nodes
   - More general
   - Requires consistency management
   - Caching provides lower latency (proportional to number of copies)
4. If access latency (performance) is a primary concern, which of the following techniques would be best to use in your DSM design?
   - Migration - No (only okay for SRSW)
   - Caching - Yes (for many "concurrent" writes, overheads may be high!)
   - Replication - Yes (for many "concurrent" writes, overheads may be high!)

DSM Design: Consistency Management

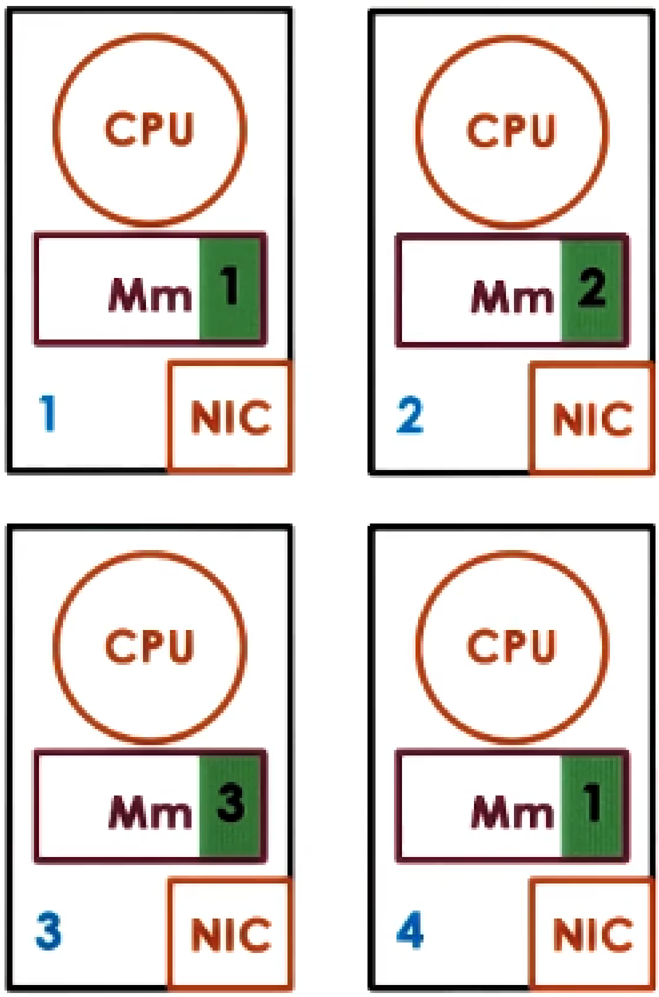
1. DSM is analogous to shared memory in shared multiprocessors
2. In SMP
   - Write invalidate - Update in one cache marks other caches as invalid
   - Write update - Update in one cache modifies other caches
   - Coherence operations triggered on each write (overhead too high)
3. In DSM
   - Push invalidations when data is written to...
     - Proactive/eager/pessimistic
     - Expect that updated state is needed immediately
   - Pull modifications periodically...
     - On demand (reactive/lazy/optimistic)
     - Expect that updated state is not needed immediately
   - These methods get triggered depending on the consistency model for the shared state

DSM Architecture

1. Page-based, OS-supported
   - Distributed nodes, each with own local memory contribution
   - Pool of pages from all nodes
   - Each page has ID, page frame number
2. If MRMW...
   - Need local caches for performance (latency)
   - Home (or manager) node drives coherence operations
   - All nodes responsible for part of distributed memory (state) management

- Each node contributes part of memory pages to DSM
- Home node manages accesses and tracks page ownership

3. "Home" node
   - Keeps state: pages accessed, modifications, caching enabled/disabled, locked…
   - Current "owner" (owner may not be home node)
4. Explicit replicas
   - Created for load balancing, performance, or reliability
   - Home/manager node controls management
   - Data centers triplicate shared state (original machine, nearby machine (same rack), remote machine (different rack or data center)



DSM Architecture

Summarizing DSM Architecture

1. Page-based DSM
   - Each node contributes part of memory pages to DSM
   - Need local caches for performance (latency)
   - All nodes responsible for part of distributed memory
   - Home node manages accesses and tracks page ownership
   - Explicit replication possible for load balancing, performance, or reliability

Indexing Distributed State

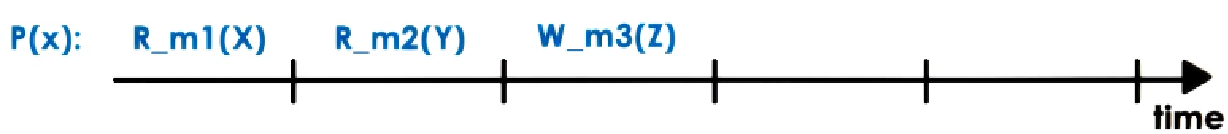
1. DSM Metadata
   - Address == node ID + page frame number
   - Node ID == "home" node
2. Global map (replicated)
   - Object (page) ID -> manager node ID
3. Global mapping table
   - Object ID -> index into mapping table -> manager node
4. Metadata for local pages (partitioned)
   - Per-page metadata is distributed across managers

Implementing DSM

1. Problem: DSM must "intercept" accesses to remote DSM state
   - To send remote messages requesting access
   - To trigger coherence messages
   - Overheads should be avoided for local, non-shared state (pages)
   - Dynamically "engage" and "disengage" DSM when necessary
2. Solution: Use hardware MMU support
   - Trap into OS if mapping invalid or access not permitted
   - Remote address mapping -> trap and pass to DSM to send message
   - Cached content -> trap and pass to DSM to perform necessary coherence operations
   - Other MMU information is useful (dirty page)

What is a Consistency Model?

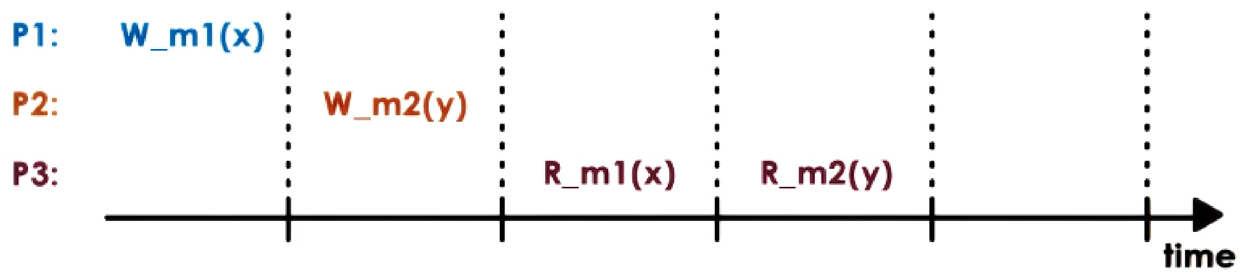
1. Consistency model == agreement between memory (state) and upper software layers
2. "Memory behaves correctly if and only if software follows specific rules"
   - Memory (state) guarantees to behave correctly...
     - Access ordering
     - Propagation/visibility of updates
   - Software might need additional atomic operations to provide other guarantees
3. Timeline notation
   - R_m1(x) == X was read from memory location m1
   - W_m1(y) == Y was written to memory location m1
   - At t=0, all memory is set to 0



DSM Architecture

Strict Consistency

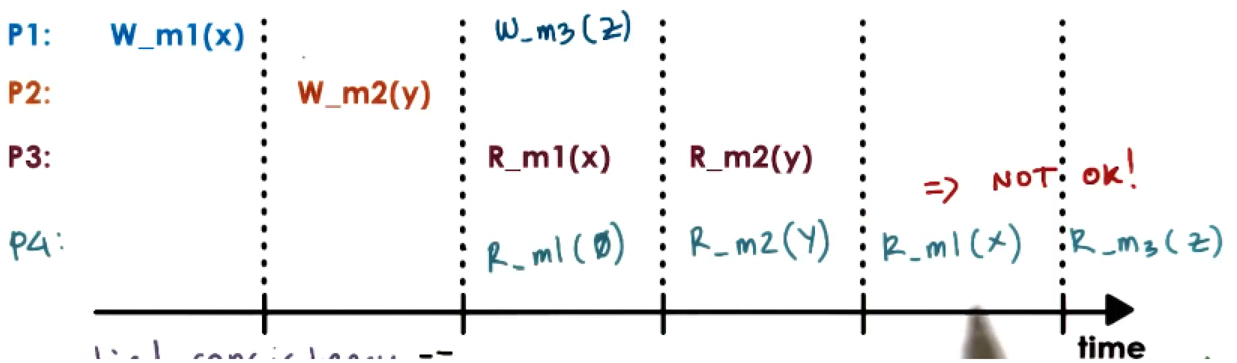
1. Strict consistency: Updates visible everywhere immediately
2. In practice, even on single SMP, no guarantees on order without extra locking and synchronization
3. In distributed systems, latency and message reorder/loss make this even harder
4. Impossible to guarantee, nice theoretical model



Strict Consistency

Sequential Consistency

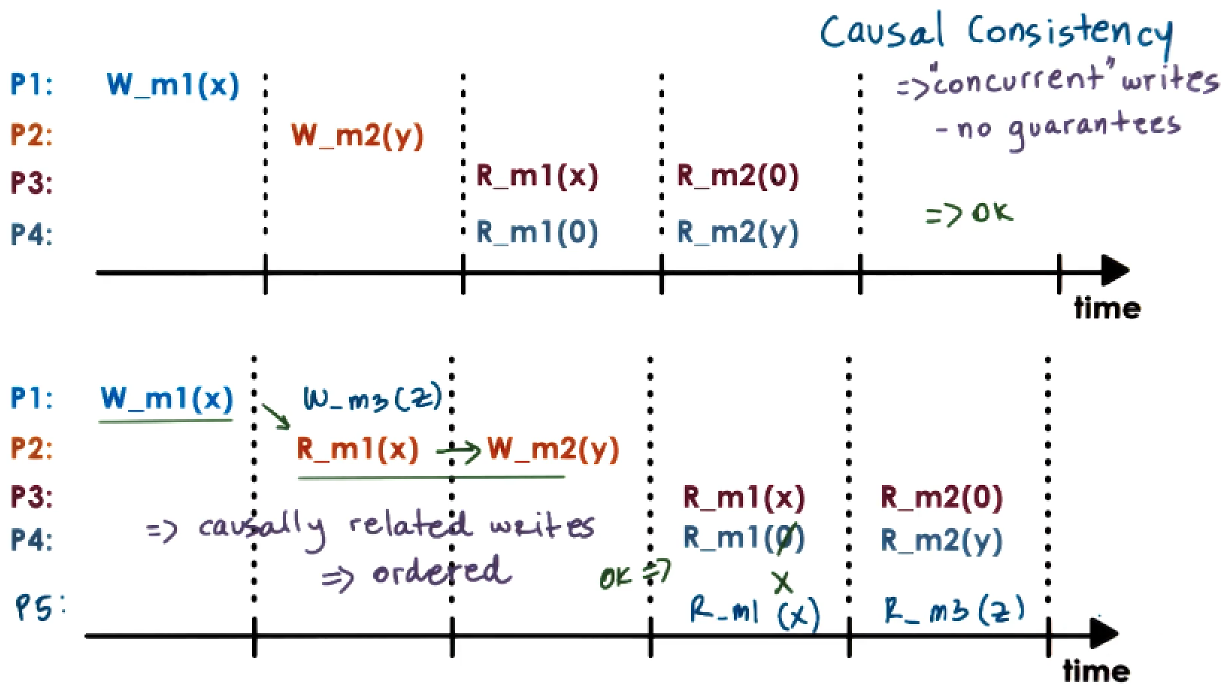
1. Memory updates from different processes may be arbitrarily interleaved
2. All process will see the same interleaving (might not be the actual order that they occurred though)



Sequential Consistency

Causal Consistency

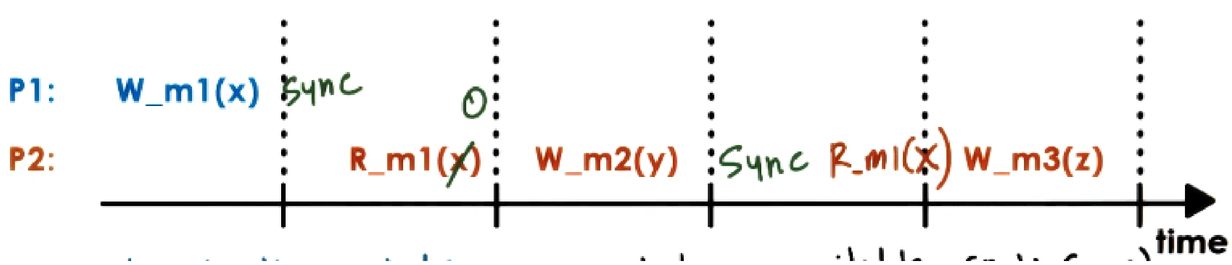
1. Software detects potential relationships between writes
   - Guarantees that the order of these relationships will be correct
2. Does not permit writes from a single process to be arbitrarily reordered
3. For “concurrent” writes (not causally related), no guarantees



Causal Consistency

Weak Consistency

1. A write to a memory location isn't necessarily predicated on the read that happened prior, as causal consistency assumes
2. Weak synchronization doesn't make the same assumption
3. Instead, introduces synchronization points
4. Synchronization points: Operations that are available (R,W,sync)
    - All updates prior to a synchronization point will be visible
    - No guarantee what happens in between
    - Must be called by process performing update and processes that need to see the update
5. Variations
    - Single synchronization operation (sync)
    - Separate sync per subset of state (page)
    - Separate "entry/acquire" vs "exit/release" operations
6. Pro: Try to limit data movement and coherence operations
7. Con: Maintain extra state for additional operations



Weak Consistency