

TreadMarks

Main Takeaways

- Distributed Shared Memory
- Relaxed Consistency Model

Why Distributed Shared Memory

- collaborative or concurrent applications use two main mechanisms to communicate or synchronize:
 - message passing (send/receive) or shared memory
- to port these apps to systems without physical shared memory, you need support for DSM
 - need this for scaling across multiple disjoint systems/nodes (supercomputers, MPP are too costly)
 - even some many core systems may not have uniform memory access from all cores
 - need to take advantage of RDMA mechanisms

DSM

- objectives: *minimize latency* and *keep coherent*
- software presents the abstraction of shared memory
- either an OS or a language run-time manages the shared memory
- there are many different DSM granularities
- page-based DSM: like regular virtual memory (e.g., Clouds here at Tech!)
- shared-variable DSM and object-based DSM: managed by a language run-time system

Implementation

- Think of how you would implement such a “virtual” shared memory among workstations
 - Software: at the OS level for unsuspecting processes
 - how can this be done efficiently?
 - Software: at a language runtime level (e.g., Java VM)
 - how can it be done without hardware support?
 - Hardware level support
 - most efficient, but costly; many exotic interconnects have some support (Quadrics, Infiniband...)
 - Hybrid

Granularity of sharing

- cache/bus line -> overkill in distributed environments
- page-based
- object-based -> typically language/runtime support

Granularity trade-offs

- finer granularity
 - improve concurrency, increase communication and frequency of execution of consistency related protocols
- coarser granularity
 - limit concurrency (especially is single writer only), reduce comm/consistency protocols
 - issue with false sharing
 - may be able to reduce with careful layout of data structures,
 - hard if implemented at system/hardware level

Objective: Reduce Latency

- Approach:
 - migration
 - migrate shared unit (page, object, etc.) to node which has current access -> need state to determine current location
 - replication
 - keep multiple copies -> need per shared unit state for keeping track of replicas, and types of access at each replica

Access Algorithm

- Single Reader – Single Writer
 - simplest, migration can suffice, not very efficient
- Multiple Readers – Single Writer
 - more general, need to track current/most recent writer (aka owner)
- Multiple Readers – Multiple Writers
 - maximum concurrency, need to resolve write conflicts through consistency protocols

TreadMarks

- Page-based
- API lets applications specify which memory is shared
- Multiple readers-multiple writers
- Does not force sequential consistency (overkill for many apps)
- Lazy consistency model: Release Consistency

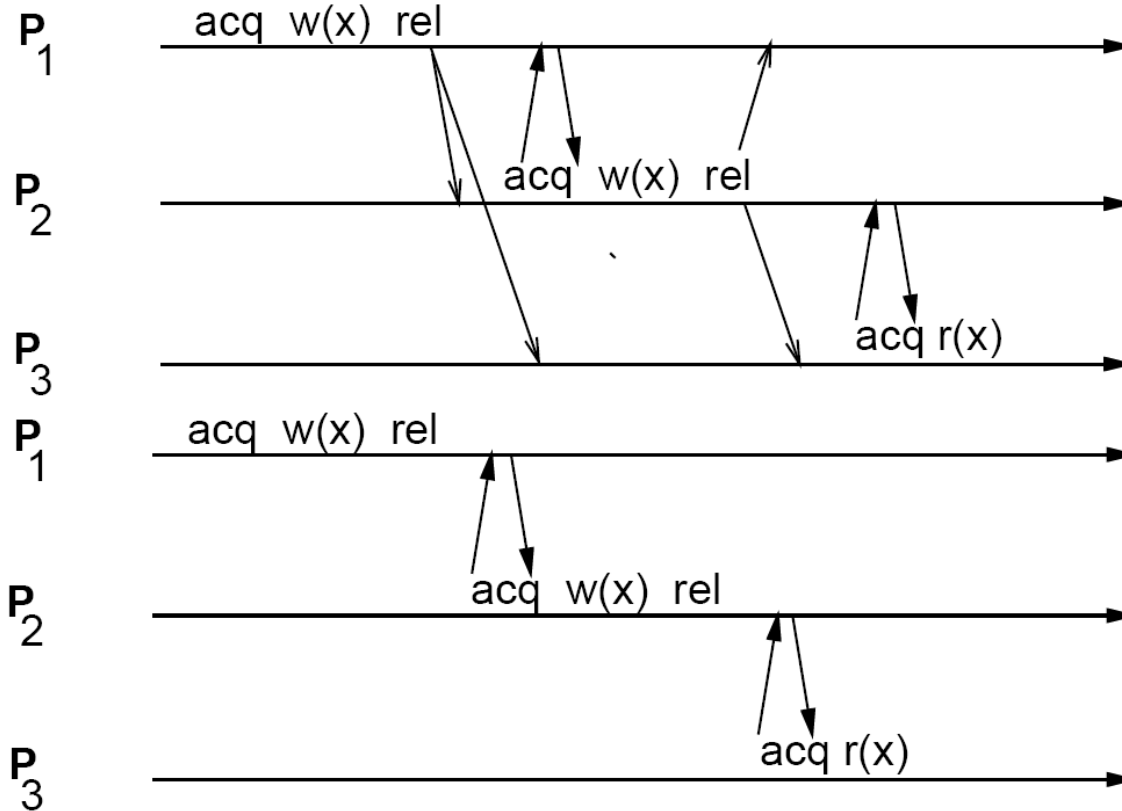
Implementation details

- Implemented as a user level lib
- Shared regions – mprotect -> SIGSEGV
 - For write bcopy to create a twin of page
- Messages raise SIGIO, select() to choose which socket to handle

Lazy Consistency – Basic Idea

- Consistency model must invalidate data/cached copies
- Lazy – invalidate just when you need to
 - e.g., invalidate state protected by lock only in cache of next proc to acquire lock
 - e.g., invalidate state updated in an iteration before barrier, only after all processes reach & leave barrier
 - NOTE: must provide hooks to tell DSM system “WHEN” to synchronize (Release Consistency)

Lazy vs. Eager



- Lazy Release Consistency: on acquire must synchronize
- (Eager Release Consistency: on release)
- TreadMarks knows that there is a lock, not what exactly is protected by lock, so will synch all modified shared state!!!
- (Entry Consistency: sync objects associated with each shared data object, i.e., invalidate not just WHEN you need to, but also only WHAT you need to)

Dealing with multiple writers

- E.g., within an iteration protected by barrier
- For each writer, create a diff
- Then merge/apply diffs to original copy
- If there were data races in sequential code, they'll still exist in TreadMarks