Motivation

**SMP systems**
- Run parts of a program in parallel
- Share single address space
  - Share data in that space
- Use threads for parallelism
- Use synchronization primitives to prevent race conditions

**Can we achieve this with multicomputers?**
- All communication and synchronization must be done with messages

Distributed Shared Memory (DSM)

**Goal:** allow networked computers to share a region of virtual memory

- How do you make a distributed memory system appear local?
- Physical memory on each node used to hold pages of shared virtual address space. Processes address it like local memory.

Take advantage of the MMU

- Page table entry for a page is valid if the page is held (cached) locally
- Attempt to access non-local page leads to a page fault
- **Page fault handler**
  - Invokes DSM protocol to handle fault
  - Fault handler brings page from remote node
- Operations are transparent to programmer
  - DSM looks like any other virtual memory system

Simplest design

Each page of virtual address space exists on only one machine at a time
- **no caching**

```
On page fault:
- Consult central server to find which machine is currently holding the page
- Directory

Request the page from the current owner:
- Current owner invalidates PTE
- Sends page contents
- Recipient allocates frame, reads page, sets PTE
- Informs directory of new location
```
Problem

Directory becomes a bottleneck
- All page query requests must go to this server

Solution
- Distributed directory
- Distribute among all processors
- Each node responsible for portion of address space
- Find responsible processor:
  - hash(page#) mod num_processors

Distributed Directory

Design Considerations: granularity

- Memory blocks are typically a multiple of a node's page size to integrate with VM system
- Large pages are good
  - Cost of migration amortized over many localized accesses
- BUT
  - Increases chances that multiple objects reside in one page
    - Thrashing
      (page data ping-pongs between multiple machines)
    - False sharing
      (unrelated data happens to live on the same page, resulting in a need for the page to be shared)

Design Considerations: replication

What if we allow copies of shared pages on multiple nodes?

- Replication (caching) reduces average cost of read operations
  - Simultaneous reads can be executed locally across hosts
- Write operations become more expensive
  - Cached copies need to be invalidated or updated
- Worthwhile if reads/writes ratio is high

Replication

Multiple readers, single writer
- One host can be granted a read-write copy
- Or multiple hosts granted read-only copies

Read operation:
- If page not local
  - Acquire read-only copy of the page
  - Set access writes to read-only on any writeable copy on other nodes

Write operation:
- If page not local or no write permission
  - Revoke write permission from other writable copy (if exists)
  - Get copy of page from owner (if needed)
  - Invalidate all copies of the page at other nodes
Full replication

Extend model
- Multiple hosts have read/write access
- Need multiple-readers, multiple-writers protocol
- Access to shared data must be controlled to maintain consistency

Dealing with replication

- Keep track of copies of the page
  - Directory with single node per page not enough
  - Keep track of copyset
    - Set of all systems that requested copies

- On getting a request for a copy of a page:
  - Directory adds requestor to copyset
  - Page owner sends page contents to requestor

- On getting a request to invalidate page:
  - Directory issues invalidation requests to all nodes in copyset and wait for acknowledgements

How do you propagate changes?

- Send entire page
  - Easiest, but may be a lot of data

- Send differences
  - Local system must save original and compute differences

Home-based algorithms

Home-based
- A node (usually first writer) is chosen to be the home of the page
- On write, a non-home node will send changes to the home node
  - Other cached copies invalidated
- On read, a non-home node will get changes (or page) from home node

Non-home-based
- Node will always contact the directory to find the current owner (latest copy) and obtain page from there

Consistency Model

Definition of when modifications to data may be seen at a given processor

Defines how memory will appear to a programmer

Places restrictions on what values can be returned by a read of a memory location

Consistency Model

Must be well-understood

- Determines how a programmer reasons about the correctness of a program
- Determines what hardware and compiler optimizations may take place
Sequential Semantics

Provided by most (uniprocessor) programming languages/systems

Program order

The result of any execution is the same as if the operations of all processors were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by the program.

— Leslie Lamport

Sequential Semantics

Requirements:
- All memory operations must execute one at a time
- All operations of a single processor appear to execute in program order
- Interleaving among processors is OK

Achieving sequential semantics

Illusion is efficiently supported in uniprocessor systems
- Execute operations in program order when they are to the same location or when one controls the execution of another
- Otherwise, compiler or hardware can reorder

Compiler:
- Register allocation, code motion, loop transformation, ...

Hardware:
- Pipelining, multiple issue, VLIW, ...

Sequential semantics

Achieving sequential consistency

Processor must ensure that the previous memory operation is complete before proceeding with the next one

Program order requirement
- Determining completion of write operations
  - get acknowledgement from memory system
- If caching used
  - Write operation must send invalidate or update messages to all cached copies
  - ALL these messages must be acknowledged

Achieving sequential consistency

All writes to the same location must be visible in the same order by all processes

Write atomicity requirement
- Value of a write will not be returned by a read until all updates/invalidates are acknowledged
  - hold off on read requests until write is complete
- Totally ordered reliable multicast
Improving performance

Break rules to achieve better performance
- But compiler and/or programmer should know what's going on!

Goals:
- Combat network latency
- Reduce number of network messages

Relaxing sequential consistency
- Weak consistency models

Relaxed (weak) consistency

Relax program order between all operations to memory
- Read/writes to different memory operations can be reordered

Consider:
- Operation in critical section (shared)
- One process reading/writing
- Nobody else accessing until process leaves critical section

No need to propagate writes sequentially or at all until process leaves critical section

Synchronization variable (barrier)

- Operation for synchronizing memory
- All local writes get propagated
- All remote writes are brought in to the local processor
- Block until memory synchronized

Consistency guarantee

- Access to synchronization variables are sequentially consistent
  - All processes see them in the same order
- No access to a synchronization variable can be performed until all previous writes have completed
- No read or write permitted until all previous accesses to synchronization variables are performed
  - Memory is updated during sync

Problems with sync consistency

- Inefficiency
  - Are we synchronizing because the process finished memory accesses or is about to start?
- On a sync, systems must make sure that:
  - All locally-initiated writes have completed
  - All remote writes have been acquired

Can we do better?

Separate synchronization into two stages:
1. acquire access
   Obtain valid copies of pages
2. release access
   Send invalidations or updates for shared pages that were modified locally to nodes that have copies.

acquire(R) // start of critical section
Do stuff
release(R) // end of critical section

Eager Release Consistency (ERC)
Getting Lazy

The release operation requires:
- Sending invalidations to copyset nodes
- And waiting for all to acknowledge

Do not make modifications visible globally at release

On release:
- Send invalidation only to directory
  or send updates to home node (owner of page)

On acquire:
- This is where modifications are propagated
  - Check with directory to see whether it needs a new copy
    - Chances are not every node will need to do an acquire

Reduces message traffic on releases

[Lazy Release Consistency (LRC)]

Finer granularity

Release consistency
- Synchronizes all data
- No relation between lock and data

Use object granularity instead of page granularity
- Each variable or group of variables can have a synchronization variable
- Propagate only writes performed in those sections
- Cannot rely on OS and MMU anymore
  - Need smart compilers

[Entry Consistency]

The end.