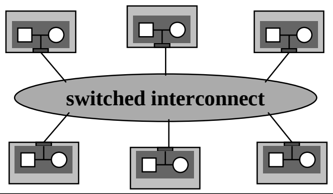
Distributed Memory and Cache Consistency

(some slides courtesy of Alvin Lebeck)

Software DSM 101

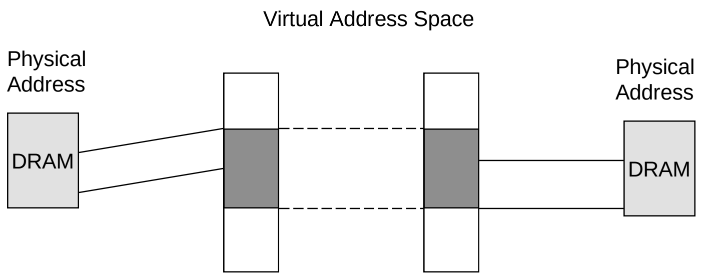
Software-based distributed shared memory (DSM) provides an illusion of shared memory on a cluster.

- remote-fork the same program on each node
- data resides in common virtual address space
  - library/kernel collude to make the shared VAS appear consistent
- The Great War: shared memory vs. message passing
  - for the full story, take Alvin Lebeck's parallel architecture class



Page Based DSM (Shared Virtual Memory)

Virtual address space is shared



Inside Page-Based DSM (SVM)

The page-based approach uses a write-ownership token protocol on virtual memory pages.

- Kai Li [Ivy, 1986], Paul Leach [Apollo, 1982]
- System maintains per-node per-page access mode.
  - {shared, exclusive, no-access}
  - determines local accesses allowed
  - modes enforced with VM page protection

mode	load	store
shared	yes	no
exclusive	yes	yes
no-access	no	no

The SVM Protocol

- Any node with access has the *latest* copy of the page.
  - On any transition from **no-access**, fetch copy of page from a current holder.
- A node with **exclusive** access holds the *only* copy.
  - At most one node may hold a page in **exclusive** mode.
  - On transition into **exclusive**, invalidate all remote copies and set their mode to **no-access**.
- Multiple nodes may hold a page in **shared** mode.
  - Permits concurrent reads: every holder has the same data.
  - On transition into **shared** mode, invalidate the **exclusive** remote copy (if any), and set its mode to **shared** as well.

Paged DSM/SVM Example

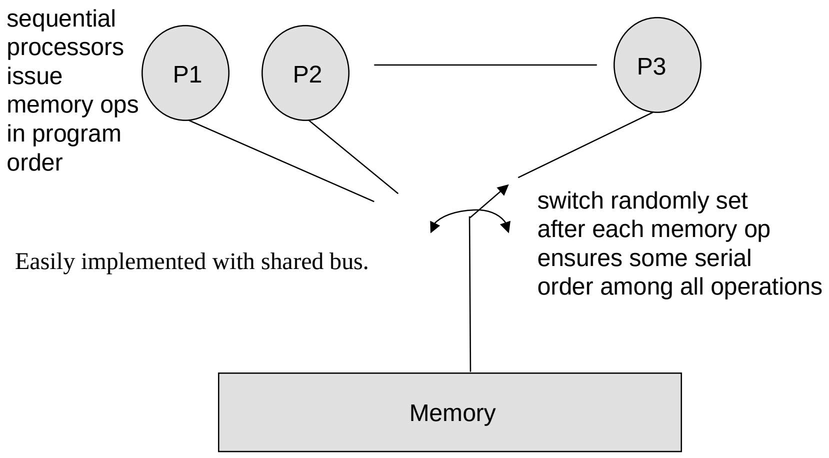
P1 read virtual address x

- Page fault
- Allocate physical frame for page(x)
- Request page(x) from home(x)
- Set readable page(x)
- Resume

P1 write virtual address x

- Protection fault
- Request exclusive ownership of page(x)
- Set writeable page(x)
- Resume

The Sequential Consistency Memory Model


sequential processors issue memory ops in program order
P1
P2
P3
switch randomly set after each memory op ensures some serial order among all operations
Memory

Easily implemented with shared bus.

Motivation for Weaker Orderings

1. Sequential consistency allows shared-memory parallel computations to execute correctly.
2. Sequential consistency is slow for paged DSM systems.
   - Processors cannot observe memory bus traffic in other nodes.
   - Even if they could, no shared bus to serialize accesses.
   - Protection granularity (pages) is too coarse.
3. Basic problem: the need for exclusive access to cache lines (pages) leads to *false sharing*.
   - Causes a “ping-pong effect” if multiple writers to the same page.
4. Solution: allow *multiple writers* to a page if their writes are “nonconflicting”.

Weak Ordering

Classify memory operations as *data* or *synchronization*

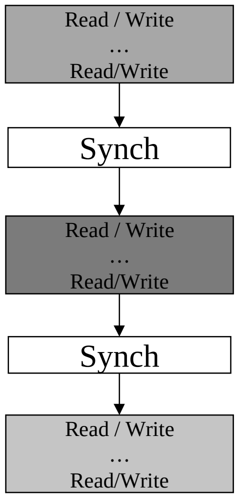
Can reorder data operations between synchronization operations

Forces consistent view at all synchronization points

Visible synchronization operation, can flush write buffer and obtain ACKS for all previous memory operations

Cannot let synch operation complete until previous operations complete (e.g., ACK all invalidations)

Weak Ordering Example



```
A                                B
                                 (x = y = 0;)
if (y > x)                       loop {
   panic("ouch");                   x = x + 1;
                                    y = y + 1;
                                 }
```

```
A                          B
acquire();                 loop() {
if (y > x)                    acquire();
   panic("ouch");             x = x + 1;
release();                    y = y + 1;
                              release();
                           }
```

Multiple Writer Protocol

x & y on same page P1 writes x, P2 writes y

Don't want delays associated with constraint of exclusive access

Allow each processor to modify its local copy of a page between synchronization points

Make things consistent at synchronization point

Treadmarks 101

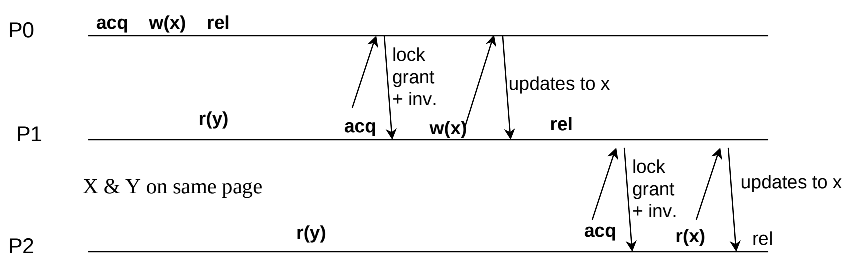
Goal: implement the "laziest" software DSM system.

- Eliminate false sharing by *multiple-writer protocol*.
  - Capture page updates at a fine grain by "diffing".
  - Propagate just the modified bytes (deltas).
  - Allows merging of concurrent nonconflicting updates.
- Propagate updates only when needed, i.e., when program uses shared locks to force consistency.
  - Assume program is *fully synchronized*.
- *lazy release consistency* (LRC)
  - *A* need not be aware of *B*'s updates except when needed to preserve potential causality...
  - ...with respect to shared synchronization accesses.

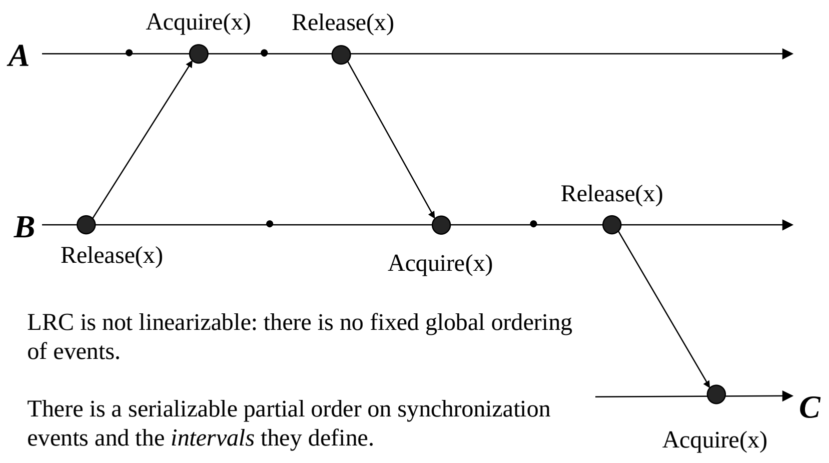
Lazy Release Consistency

Piggyback write notices with *acquire* operations.

- guarantee updates are visible on acquire, not release
  lazier than Munin
- implementation propagates invalidations rather than updates



Ordering of Events in Treadmarks



LRC is not linearizable: there is no fixed global ordering of events.

There is a serializable partial order on synchronization events and the *intervals* they define.

Vector Timestamps in Treadmarks

To maintain the partial order on intervals, each node maintains a *current vector timestamp* (CVT).

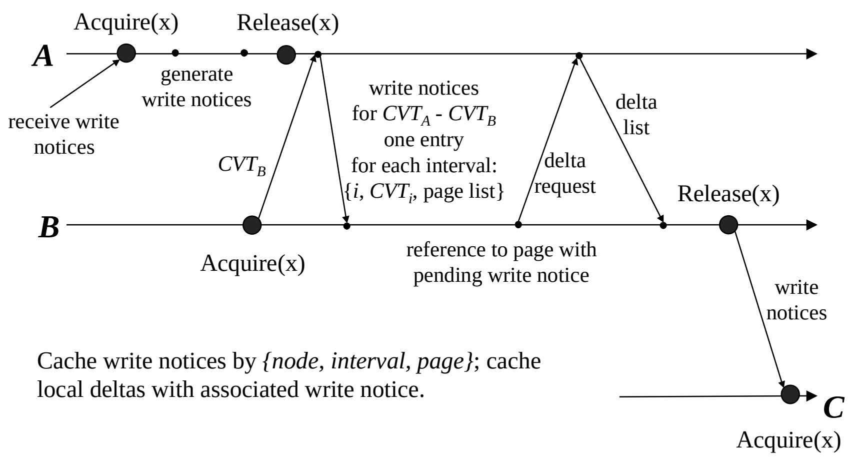
- Intervals on each node are numbered 0, 1, 2...
- *CVT* is a vector of length *N*, the number of nodes.
- *CVT[i]* is number of the last *preceding* interval on node *i*.

Vector timestamps are updated on lock *acquire*.

- *CVT* is passed with lock acquire request...
- compared with the holder's *CVT*...
- pairwise maximum *CVT* is returned with the lock.

Systems & Architecture

LRC Protocol



Cache write notices by *{node, interval, page}*; cache local deltas with associated write notice.

Systems & Architecture

Write Notices

LRC requires that each node be aware of any updates to a shared page made during a preceding interval.

- Updates are tracked as sets of *write notices*.
  - A *write notice* is a record that a page was dirtied during an interval.
- Write notices propagate with locks.
  - When relinquishing a lock token, the holder returns all write notices for intervals “added” to the caller’s *CVT*.
- Use page protections to collect and process write notices.
  - “First” store to each page is trapped...write notice created.
  - Pages for received write notices are invalidated on acquire.

Capturing Updates (Write Collection)

To permit multiple writers to a page, updates are captured as deltas, made by “diffing” the page.

- Delta records include only the bytes modified during the interval(s) in question.
- On “first” write, make a copy of the page (a *twin*).
  - Mark the page **dirty** and write-enable the page.
  - Send write notices for all dirty pages.
- To create deltas, diff the page with its twin.
  - Record deltas, mark page **clean**, and disable writes.
- Cache write notices by *{node, interval, page}*; cache local deltas with associated write notice.

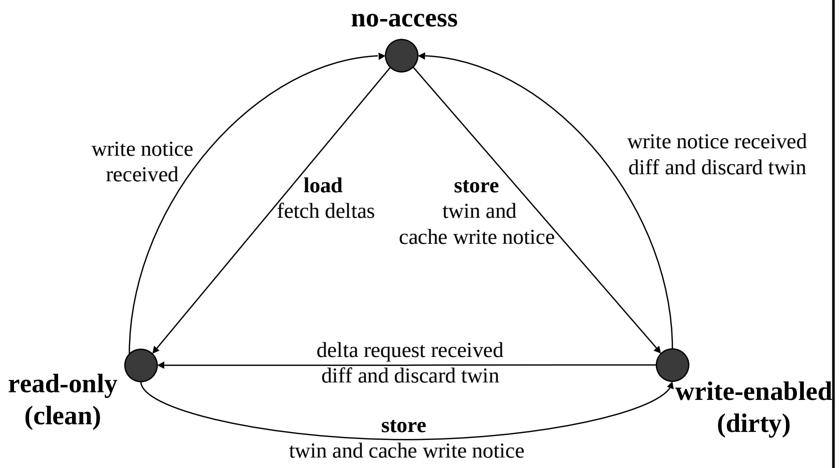
Lazy Interval/Diff Creation

1. Don’t create intervals on every acquire/release; do it only if there’s communication with another node.
2. Delay generation of deltas (diff) until somebody asks.
   - When passing a lock token, send write notices for modified pages, but leave them write-enabled.
   - Diff and mark clean if somebody asks for deltas.

     Deltas may include updates from later intervals (e.g., under the scope of other locks).
3. Must also generate deltas if a write notice arrives.

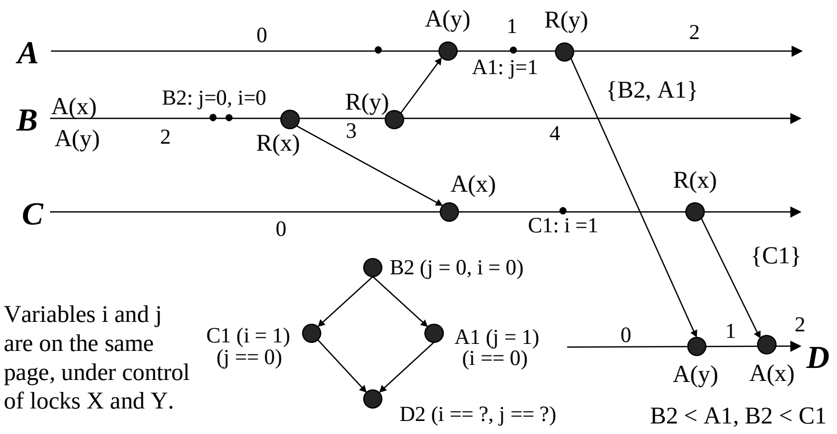
   Must distinguish local updates from updates made by peers.
4. Periodic garbage collection is needed.

Treadmarks Page State Transitions



Ordering Conflicting Updates

Write notices must include origin node and CVT.
Compare CVTs to order the updates.



Variables i and j are on the same page, under control of locks X and Y.

Ordering Conflicting Updates (2)

D receives B’s write notice for the page from A.

D receives write notices for the same page from A and C, covering their updates to the page.

If D then touches the page, it must fetch updates (deltas) from three different nodes (A, B, C), since it has a write notice from each of them.

The deltas sent by A and B will both include values for j.

The deltas sent by B and C will both include values for i.

D must decide whose update to j happened first: B’s or A’s.

D must decide whose update to i happened first: B’s or C’s.

In other words, D must decide which order to apply the three deltas to its copy of the page.

D must apply these updates in vector timestamp order.

Every write notice (and delta) must be tagged with a vector timestamp.

Page Based DSM: Pros and Cons

Good things

Low Cost, can use commodity parts

Flexible Protocol (Software)

Allocate/replicate in main memory

Bad Things

Access Control Granularity

- False sharing
  - Complex protocols to deal with false sharing

Page fault overhead

DUKE Systems & Architecture

Another Peak at File Cache Consistency

Today's software DSMs can efficiently keep a shared data space consistent under certain assumptions.

- peer-peer: clients are mutually trusting
- data may become unavailable if client fails
- fine-grained synchronization visible to cache manager

Distributed file caches operate with different assumptions:

- clients trust only the server (unless they share files)
  - except *cooperative caching* (e.g., xFS or GMS)
- must be resilient to client failures
- no fine-grained synchronization

DUKE Systems & Architecture

File Cache Example: NQ-NFS Leases

In NQ-NFS, a client obtains a *lease* on the file that permits the client's desired read/write activity.

"A lease is a ticket permitting an activity; the lease is valid until some expiration time."

- A *read-caching lease* allows the client to cache clean data.

  Guarantee: no other client is modifying the file.

- A *write-caching lease* allows the client to buffer modified data for the file.

  Guarantee: no other client has the file cached.

Leases may be revoked by the server if another client requests a conflicting operation (server sends *eviction notice*).

Systems & Architecture

Using NQ-NFS Leases

1. When a client begins to read/write a file, the server issues an appropriate lease to the client.

   read leases may include multiple clients

   write leases are issued exclusively to one client

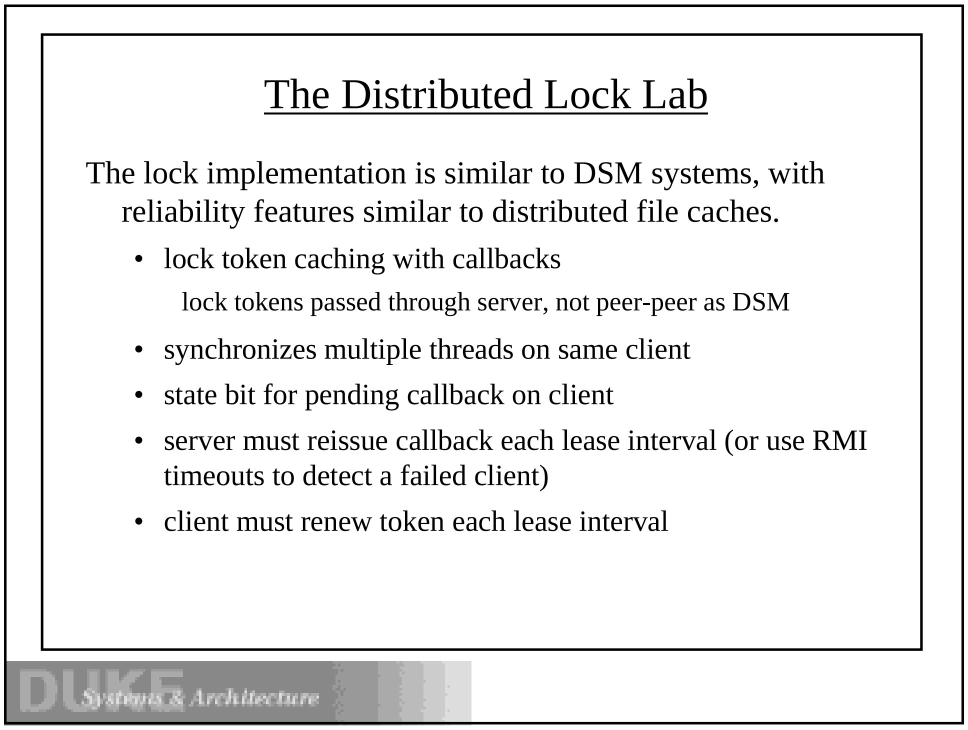
2. Before the lease expires, the client must *renew* the lease.
3. If a client is evicted from a write lease, it must writeback.

   server grants lease extensions as long as the client writes

   client sends "vacated" notice when all writes complete

4. If a client fails, the server reclaims the lease.
5. If the server fails and recovers, it must wait for one lease period before issuing new leases.

Systems & Architecture



The Distributed Lock Lab

The lock implementation is similar to DSM systems, with reliability features similar to distributed file caches.

- lock token caching with callbacks
  - lock tokens passed through server, not peer-peer as DSM
- synchronizes multiple threads on same client
- state bit for pending callback on client
- server must reissue callback each lease interval (or use RMI timeouts to detect a failed client)
- client must renew token each lease interval