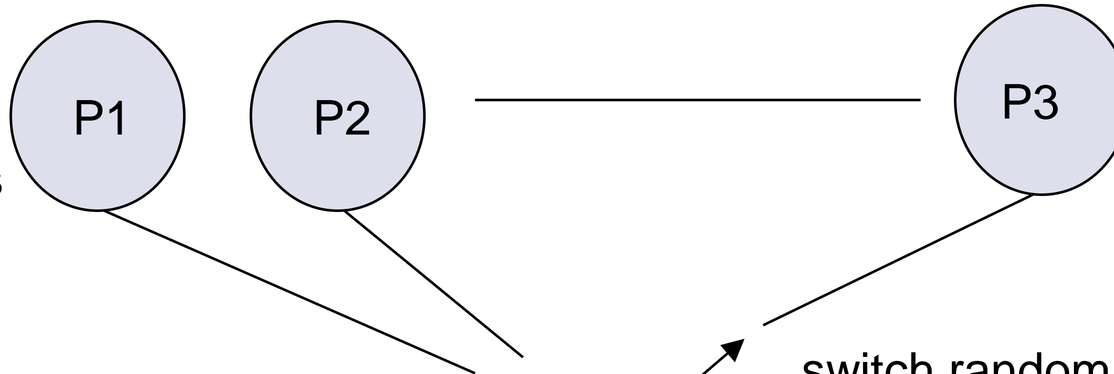
Multiple-Writer Distributed Memory

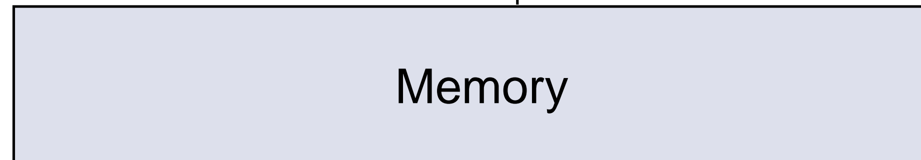
The Sequential Consistency Memory Model

sequential processors issue memory ops in program order



switch randomly set after each memory op ensures some serial order among all operations

Easily implemented with shared bus.



[Lebeck]

This Class

Spend time at the start to discuss Haifeng Yu's notion of a continuum of consistency quality for distributed systems.

Understand why weakening consistency can improve performance or availability.

But what does it really mean to weaken consistency? How much can the application tolerate?

We discuss one notion of weak consistency for distributed shared memory (DSM), using mechanisms similar to Bayou. The background for DSM was covered previously and is in the CDK text chapter 16.

Motivation for Weaker Orderings

1. Sequential consistency is sufficient (but not necessary) for 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

Careful access ordering only matters when data is shared.

Shared data should be synchronized.

Classify memory operations as *data* or *synchronization*

Can reorder data operations between synchronization operations

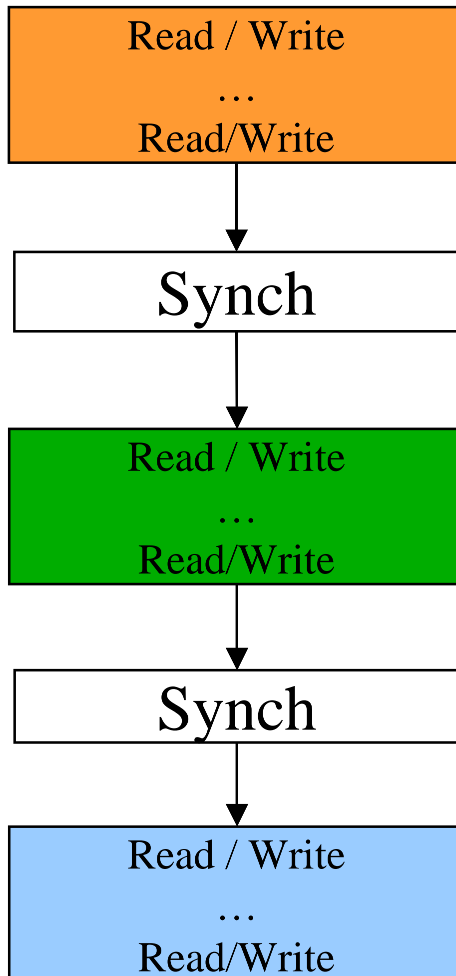
Forces consistent view at all synchronization points

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)

[Lebeck]

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();
                           }
```


[Lebeck]

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

[Lebeck]

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.

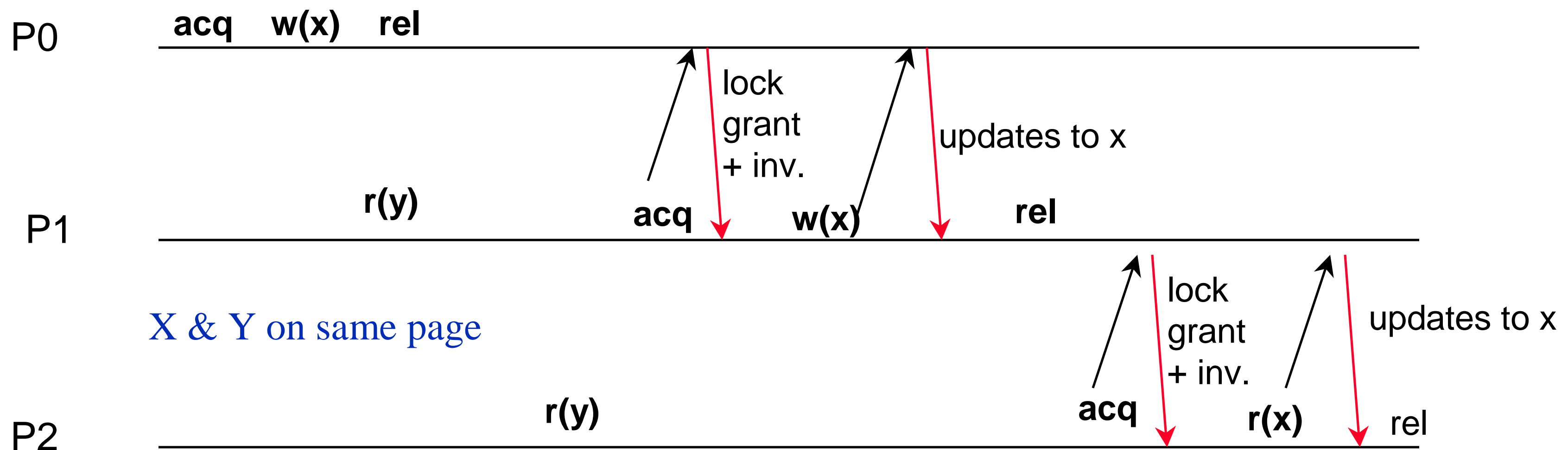
Lazy Release Consistency

Piggyback write notices with *acquire* operations.

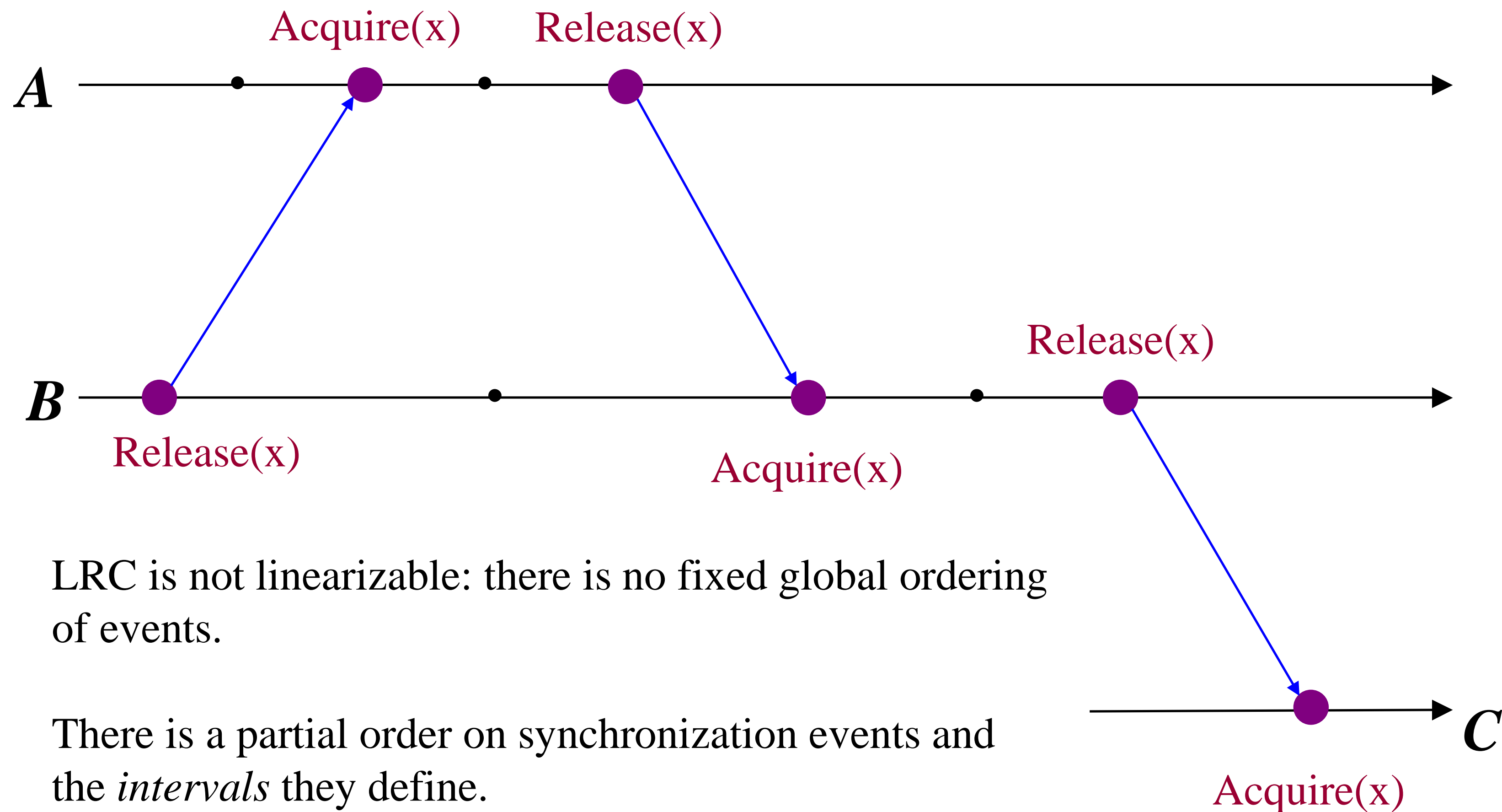
- guarantee updates are visible on acquire

  lazier than Munin, which propagates updates on release

- 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 partial order on synchronization events and the *intervals* they define.

Vector Timestamps in Treadmarks

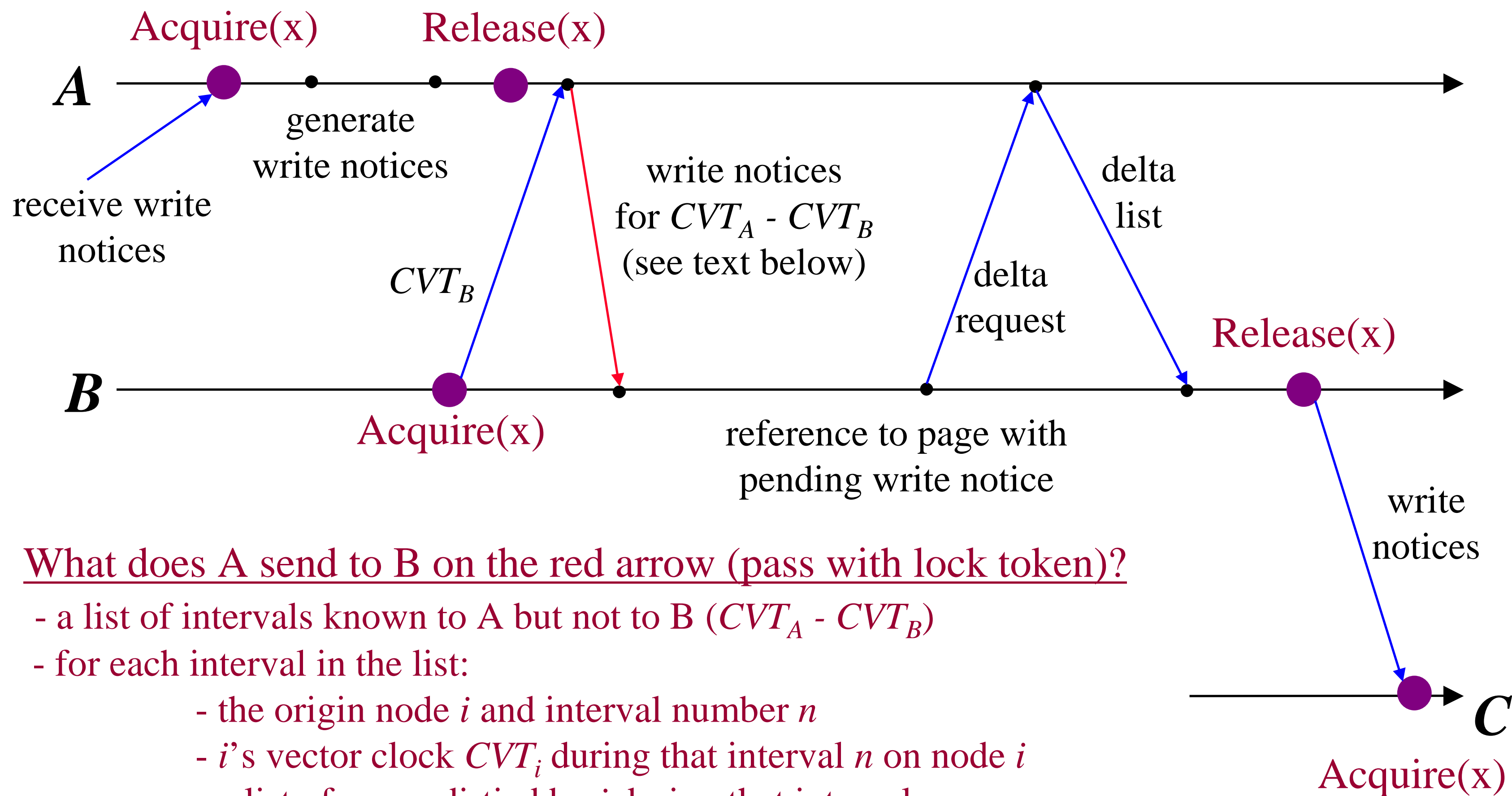
To maintain the partial order on intervals, each node maintains a *current vector timestamp* (CVT).

- 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.

LRC Protocol



What does A send to B on the red arrow (pass with lock token)?

- a list of intervals known to A but not to B (CVT_A - CVT_B)
- for each interval in the list:
  - the origin node *i* and interval number *n*
  - *i*'s vector clock CVT_i during that interval *n* on node *i*
  - a list of pages dirtied by *i* during that interval *n*
  - these dirty page notifications are called *write notices*

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*.

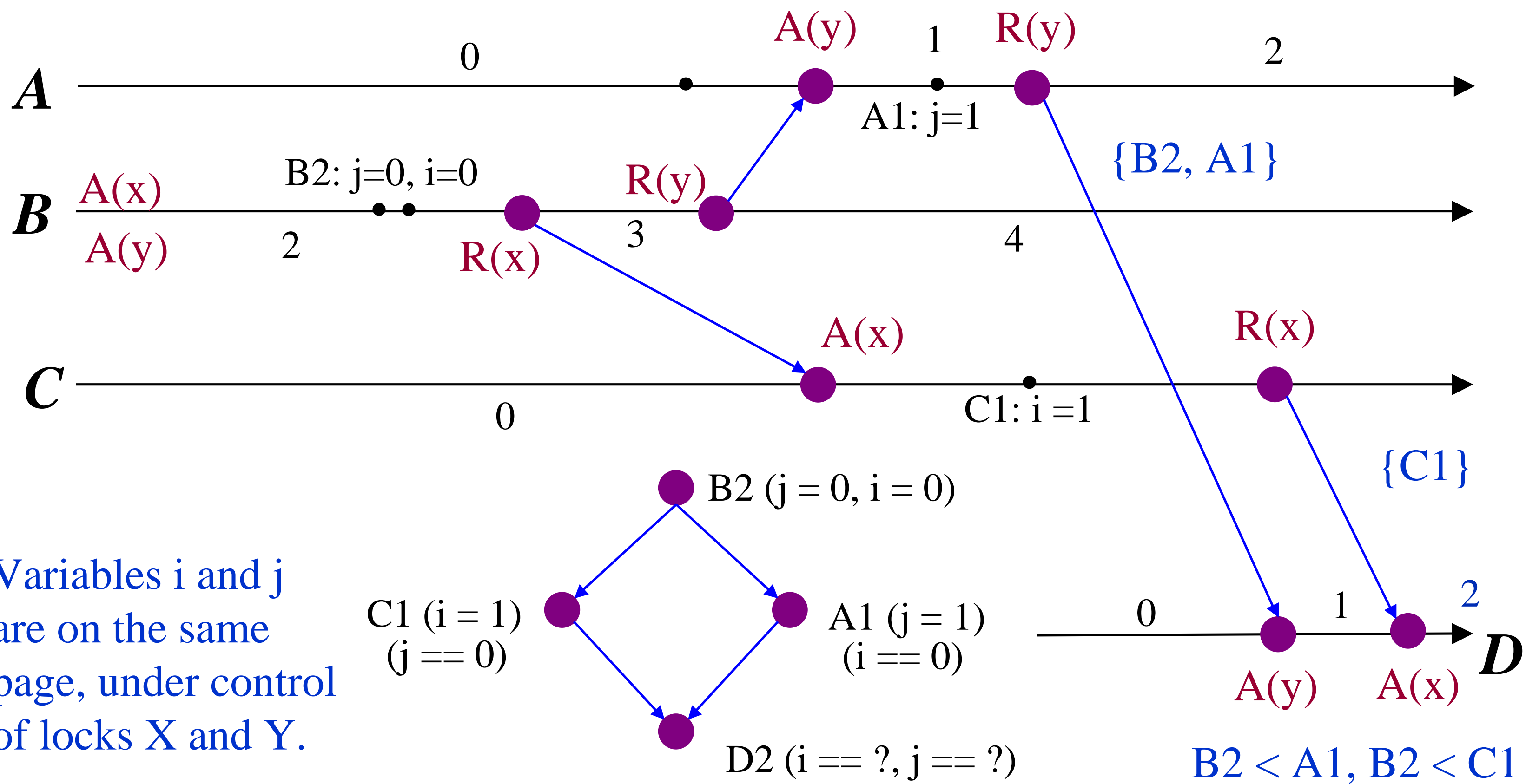
- Use page protections to collect and process write notices.

  "First" store to each page is trapped...write notice created.

  Pages for received write notices are invalidated on acquire.

Ordering Conflicting Updates

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



More slides

The following slides were not discussed in class and will not be covered on the exam. However, you should be sure that you understand the basic techniques used in software DSM systems.

You should also think about whether this is the "weakest" memory consistency we can get away with for fully synchronized programs. What if the system knew the assignment of locks to data? Can it infer this association on the fly?

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.

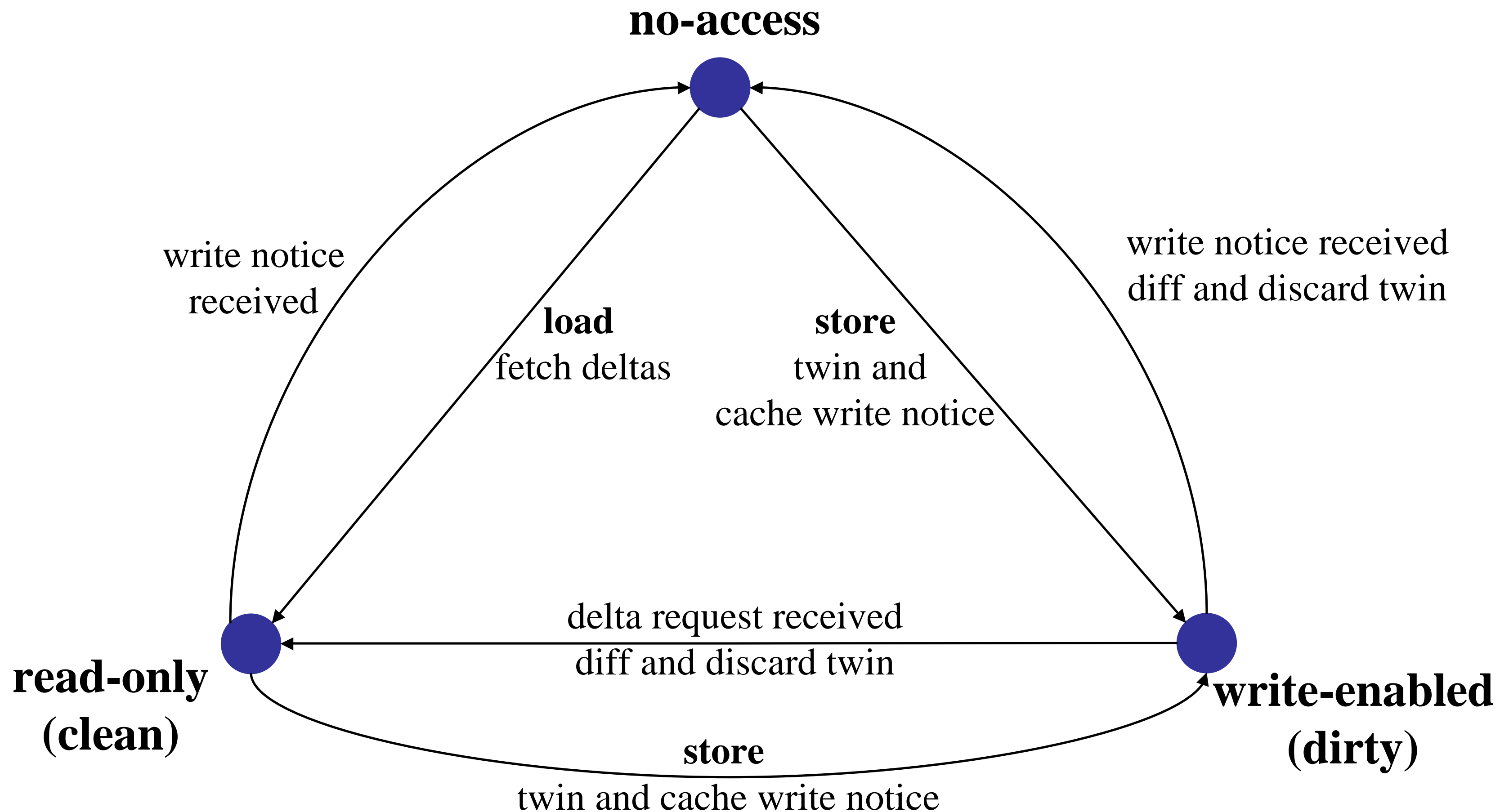
Lazy Interval/Diff Creation

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 (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