Data-Centric Consistency Models

- The general organization of a logical data store, physically distributed and replicated across multiple processes.
Consistency models

• The scenario we will be studying:
  – Some sort of shared data that we will call **data store** (examples: shared memory, shared file system)
  – multiple processes perform read/write operations on data store
  – Each process has a local (nearby copy of the entire store)

• A **consistency model** is a contract between processes and the data store
Sequential Consistency

“A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” [Lamport, 1979]

- Total order achieved by *interleaving* accesses from different processes
  - *as if* there were no replicas, and a single copy of data

- Important points:
  - Maintains *program order*
  - operations appear to execute in the *same order* to all processes

- Programmer’s intuition is maintained
Sequential Consistency example

a) A sequentially consistent data store.

b) A data store that is not sequentially consistent.
Causal Consistency (1)

• Definition:
  “Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

• C.C. is a weaker consistency model than S.C.
  – like in logical clocks, causality is invoked as the minimum logical constraint that must hold between events
### Causal Consistency (2)

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
<th>W(x)c</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: R(x)a</td>
<td>W(x)b</td>
</tr>
<tr>
<td>P3: R(x)a</td>
<td>R(x)c</td>
</tr>
<tr>
<td>P4: R(x)a</td>
<td>R(x)b</td>
</tr>
</tbody>
</table>

- This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.
## Causal Consistency (3)

**P1:** $W(x)a$

<table>
<thead>
<tr>
<th>P2:</th>
<th>R(x)a</th>
<th>W(x)b</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4:</td>
<td>R(x)a</td>
<td>R(x)b</td>
</tr>
</tbody>
</table>

(a)

**P1:** $W(x)a$

<table>
<thead>
<tr>
<th>P2:</th>
<th></th>
<th>W(x)b</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3:</td>
<td>R(x)b</td>
<td>R(x)a</td>
</tr>
<tr>
<td>P4:</td>
<td>R(x)a</td>
<td>R(x)b</td>
</tr>
</tbody>
</table>

(b)

a) A violation of a causally-consistent store.

b) A correct sequence of events in a causally-consistent store.
Weak Consistency

- Another approach to data consistency is to use synchronization primitives
  - think of these primitives as barriers, added by the programmer when enforcing of consistency is desired

- Simplest model: weak consistency
  - Single synchronization primitive with following properties:
    - Accesses to synchronization variables associated with a data store are sequentially consistent => *all processes see operations on same order*
    - No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere => *synch to flush the pipeline*
    - No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed => *synch to get the latest value*
Weak Consistency (3)

a) A valid sequence of events for weak consistency.

\[
\begin{array}{ccc}
P1: & W(x)a & W(x)b & S \\
P2: & & R(x)a & R(x)b & S \\
P3: & & R(x)b & R(x)a & S \\
\end{array}
\]

(b) An invalid sequence for weak consistency.

\[
\begin{array}{ccc}
P1: & W(x)a & W(x)b & S \\
P2: & & S & R(x)a \\
\end{array}
\]
Release Consistency (1)

- In this model the programmer has to use two separate synchronization operations
  - *Acquire* operation is used to enter a critical region
  - *Release* is used to leave the region
- This is a weaker model because the responsibility of maintaining consistency is delegated to the user
  - Acquire and release do not have to apply to all shared data in a store
  - if all data is protected, the result of the execution will be no different than with sequential consistency model
- Difference with respect to weak consistency: two separate synch operations for entering or leaving the critical section allow for potentially more efficient implementation
Release Consistency (2)

- Semantic of acquire
  - All local copies of the protected data are brought up to date with remote ones

- Semantic of release
  - All changes to local copy of protected data are propagated to remote copies

- Possible variants
  - Lazy release consistency: at time of release changes are not propagated. Instead, during an acquire the most recent values of data are retrieved from all possible copies holding them
Release Consistency (1)

- A valid event sequence for release consistency.
Implementing A Coherence Protocol

• Implementation depends on a number of factors:
  – Type of consistency desired
  – Type of (distributed) system

• Three different types of systems
  – Loosely coupled distributed systems
  – Hardware/Software distributed shared memory systems
  – Multiprocessor systems
Implementation Mechanisms

• Snooping based Implementation
  – Assumes a common bus
  – Possible only for multiprocessor systems

• Directory based implementation
  – Does not require a common bus or shared address space
Key Caching Terms to Review

- Cache block or cache line or memory block/line
- Cache hit and Cache miss
- Directly mapped cache
  - Suppose you have 64 memory blocks and 16 cache blocks
    - memory blocks 0, 16, 32, and 48 can only be cache block 0
    - memory blocks 1, 17, 33, and 49 can only be in cache block 1
    - and so on ....
Two Classifications of Protocols

- Where do you update on writes
  - Write-through (update local copy and main memory)
  - Write-back (update only local copy)
    - Need to update main memory during the write-back operation
    - Requires less bandwidth

- What do you do with cached data elsewhere?
  - Write-update or write-broadcast (send new values)
  - Write-invalidate (simply mark the copy unreadable)
    - Requires less bandwidth
A Multiprocessor Cache Coherence Protocol

- We will study a write-invalidate protocol for a write-back cache

- Two simple ideas
  - A cache block can be in three states:
    - Invalid
    - Shared (read only)
    - Exclusive (read/write)
  - Everyone follows what is happening on the bus
  - Let others know of your actions through the bus