Distributed Shared Memory and Memory Consistency

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University
Outline

- Introduction
- Strict Consistency
- Sequential Consistency
- Serializable Consistency
- Causal Consistency
- FIFO Consistency (Pipelined RAM)
- Processor Consistency (PRAM+)
- Weak Consistency
- Release Consistency
- Entry Consistency
Shared Memory

- Two types of memory organization in parallel and distributed systems
  - Shared memory (shared address space)
  - Distributed memory (message passing)

- The shared memory or single address space abstraction provides several advantages over the message passing (or private memory) abstraction
  - A more natural transition from uniprocessors
  - Simplifying difficult programming tasks such as data partitioning and dynamic load distribution

- For this reason, parallel systems that support shared memory are gaining wide acceptance in both technical and commercial computing
Distributed Shared Memory

- All computers share a single virtual address space
  - Pages can be physically located on any computer
  - When process accesses data in shared address space a *mapping manager* maps the request to the physical page
  - If the page is remote – block the process and fetch it
Advantages of DSM

- **Simpler abstraction**
  - Programmer does not have to worry about data movement, may be easier to implement than RPC since the address space is the same

- **Easier portability**
  - Sequential programs can in principle be run directly on DSM systems

- **Flexible communication**
  - No need for sender and receiver to exist, can join and leave DSM system without affecting the others

- **Process migration simplified**
  - One process can easily be moved to a different machine since they all share the address space
Memory Consistency

- Shared memory systems allow concurrent access to shared data
  - Concurrency may lead to unexpected results - what if the read does not return the value stored by the most recent write (write did not propagate)?

- Memory is *coherent* if the value returned by the read operation is always the value the programmer expected
  - To maintain coherency of shared data a mechanism that controls (and synchronizes) memory accesses is used
  - This mechanism only allows a restricted set of memory access orderings
  - *memory consistency model* - the set of allowable memory access orderings
Coherency vs. Consistency

- Coherence defines what values can be returned by a read
- Consistency determines when a written value will be returned by a read

- Coherence and consistency are complementary
  - Coherence concerns only one memory location
  - Consistency concerns apparent ordering for all locations
Memory Consistency

- Consistency models that require a global agreement
  - Strict Consistency
  - Sequential Consistency
  - Linearizable Consistency
  - Causal Consistency

- Consistency models with weak/no global agreement
  - FIFO Consistency (Pipelined RAM)
  - Processor Consistency (PRAM+)

- Consistency models with synchronization
  - Weak Consistency
  - Release Consistency
  - Entry Consistency
Strict Consistency

- **Definition**
  - “Any read on a data item x returns a value corresponding to the result of the most recent write on x”

- **Assumes the existence of absolute global time**
  - to define what “most recent” means

- **Hard to implement (actually not possible)**
  - $W_i(x)a$ – a write by process ‘i’ to item ‘x’ with a value of ‘a’
  - $R_i(x)b$ – a read by process ‘i’ from item ‘x’ producing the value ‘b’

<table>
<thead>
<tr>
<th>Process</th>
<th>Action</th>
<th>Process</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1:</td>
<td>$W(x)a$</td>
<td>P1:</td>
<td>$W(x)a$</td>
</tr>
<tr>
<td>P2:</td>
<td>$R(x)a$</td>
<td>P2:</td>
<td>$R(x)$NIL $R(x)a$</td>
</tr>
</tbody>
</table>

(a) **<Strictly Consistent>**

(b) **<Not Strictly Consistent>**
Strict Consistency

- **Strict order (global, total order)**
  - $W_2(x) \rightarrow R_1(x) \rightarrow R_3(x) \rightarrow R_1(x)$

![Diagram showing strict consistency and not strict consistency](image-url)
Sequential Consistency

- **Definition**
  - “The result of any execution of the operations of all processors is the same as if they were executed in some sequential order and each process’ operations are in the order of the program” by Lamport in 1979

- **Interleaving of operations does not matter, if all processes see the same ordering**

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

(a) <Sequentially Consistent>

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

(b) <Not Sequentially Consistent>
Linearizable Consistency (Linearizability)

- Definition
  - Sequential consistency, plus timestamping of each operation with finite precision
  - If the time stamp of operation 1 < timestamp of operation 2, the operation 1 should precede operation 2 in the total order

- Weaker than strict consistency, stronger than sequential consistency

- Linearizability assumes the existence of a global clock
  - However, as with strict consistency, such an assumption is not realistic for most distributed systems
  - Can linearizability be implemented for physically distributed data stores?
  - Answer: Yes. Linearizability assumes loosely synchronized clocks, that is, it assumes that several events may happen within the same time slot. Those events need to be ranked adhering to sequential consistency
Linearizability vs. Sequential Consistency

- Linear order (global order)
  - W2 → W3 → R1 → R4 → R4 → R1

- Sequential order (global order)
  - W3 → W2 → R1 → R4 → R4 → R1
Causal Consistency

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”
- Not required that all processes see events in the same order

Weakening of sequential consistency

Causally related operations
- Any read/write (or two write) operations on the same item are causally related
- Read and write operations by the same process are potentially causally related
- If \( W_i(X) \rightarrow R_j(X) \) and \( R_j(X) \rightarrow W_j(Y) \), then \( W_i(X) \rightarrow R_j(X) \rightarrow W_j(Y) \)
Causal Consistency

Example 1 (Causally Consistent)
- $W_1(X)a$, $R_2(X)a$, and $W_2(X)b$ are causally related
- $W_2(X)b$ and $W_1(X)c$ are not causally related but concurrent

<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>

Example 2
- $W_1(X)a$, $R_2(X)a$, and $W_2(X)b$ are causally related
- $W_1(X)a$ and $W_2(X)b$ are concurrent

(A) Not Causally Consistent

(B) Causally Consistent
Causal Consistency

- Causal order (global order)
  - W2 → W2 → W3 or W2 → W3 → W2

- Causal order (global order)
  - W2 → W3
FIFO (PRAM) Consistency

Definition

- “Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes”
- PRAM = pipelined RAM

Easy to implement

- With local timestamps (process ID, seq. #)
- Only required that writes from a single source must arrive in order as if they were in a pipeline

|   |   |   |   
|---|---|---|---|
| P1: | W(x)a |   |   |
| P2: | R(x)a | W(x)b | W(x)c |
| P3: |   | R(x)b | R(x)a | R(x)c |
| P4: |   | R(x)a | R(x)b | R(x)c |

<FIFO Consistent, but not causally consistent>
Processor Consistency

Definition
- Coherency on the same data item - all processes agree on the order of write operations to the same data item
- Aka PRAM+
FIFO vs. Processor Consistency

- **FIFO consistent order**
  - $W_2(x,a) \rightarrow W_2(x,b) \rightarrow W_2(y,a)$ is observed by all machines
  - $W_3(x,0) \rightarrow W_3(x,1) \rightarrow W_3(z,a)$ is observed by all machines

- **Processor consistent order**
  - $W_2(x,a) \rightarrow W_3(x,0) \rightarrow W_3(x,1) \rightarrow W_2(x,b)$ is observed by all machines
Weak Consistency

- Not all applications need to see all writes, let alone seeing them in the same order
  - The weak consistency models enforce consistency on a group of operations, as opposed to individual reads and writes (as is the case with strict, sequential, causal and FIFO consistency)

- The three properties of Weak Consistency:
  - P1. Accesses to synchronization variables associated with a data-store are sequentially consistent
  - P2. No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
  - P3. No data access is allowed to be performed until all previous operations to synchronization variables have been performed
Weak Consistency

By doing a sync.,

- a process can force the just written value out to all the other replicas.
- also, a process can be sure it’s getting the most recently written value before it reads

(a) A valid sequence of events for weak consistency. This is because P2 and P3 have yet to synchronize, so there’s no guarantees about the value in ‘x’

(b) An invalid sequence for weak consistency. P2 has synchronized, so it cannot read ‘a’ from ‘x’ – it should be getting ‘b’
Release Consistency

How does a weakly consistent data-store know that the sync is the result of a read or a write?
- Two sync variables can be used, "acquire" and "release", and their use leads to the "Release Consistency" model

Definition
- When a process does an "acquire", the data-store will ensure that all the local copies of the protected data are brought up to date to be consistent with the remote ones if needed:
  - Tell the memory system that a critical section is about to be entered
- When a "release" is done, protected data that have been changed are propagated out to the local copies of the data-store:
  - Tell the memory system that a critical section has just been exited
A valid event sequence for release consistency

- Process P3 has not performed an acquire, so there are no guarantees that the read of ‘x’ is consistent. The data-store is simply not obligated to provide the correct answer.
- P2 does perform an acquire, so its read of ‘x’ is consistent.

<table>
<thead>
<tr>
<th>Event Sequence</th>
<th>P1: Acq(L) W(x)a W(x)b Rel(L)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P2: Acq(L) R(x)b Rel(L)</td>
</tr>
<tr>
<td></td>
<td>P3: R(x)a</td>
</tr>
</tbody>
</table>
Eager Release vs. Lazy Release

- **Eager Release Consistency**
  - Make modifications globally visible at the time of a release

- **Lazy Release Consistency**
  - Only a processor that acquires a lock will see all modifications that precede the lock acquire
Eager Release vs. Lazy Release

Repeated Updates of Cached Copies in Eager RC.

Message Traffic in LRC.
Entry Consistency

- A different twist on things is “Entry Consistency”
  - Acquire and release are still used

- Each variable is required to be associated with some type of synchronization variable such as a lock or a barrier
  - Essentially the synchronization is individualized

- When an acquire is done on a synchronization variable, only those ordinary shared variables guarded by that synchronization variable are made consistent
  - Entry consistency differs from lazy release consistency in that the latter does not associate shared variable with locks or barriers and at acquire time has to determine empirically which variables it needs
Entry Consistency

- Locks associated with individual data items, as opposed to the entire data-store

<table>
<thead>
<tr>
<th>P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: Acq(Lx) R(x)a R(y)NIL</td>
</tr>
<tr>
<td>P3: Acq(Ly) R(y)b</td>
</tr>
</tbody>
</table>

- Note: P2’s read on ‘y’ returns NIL as no locks have been requested
# Summary

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
<th>Event Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict Consistency</td>
<td>All processes see all shared accesses in a absolute time order</td>
<td>Absolute global order</td>
</tr>
<tr>
<td>Sequential Consistency</td>
<td>All processes see all shared accesses in the same order</td>
<td>Some sequential global order + programmed local order</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes see all shared accesses in an almost absolute time order</td>
<td>Some sequential global order + programmed local order + timestamp with finite precision</td>
</tr>
<tr>
<td>Causal Consistency</td>
<td>All processes see all causally-related writes in the same order but may see concurrent writes in a different order</td>
<td>Sequential global order of causally related operations + programmed local order</td>
</tr>
<tr>
<td>FIFO Consistency</td>
<td>All processes see writes from each processor in the order they were issued</td>
<td>No global order + programmed local order</td>
</tr>
<tr>
<td>Processor Consistency</td>
<td>All processes see writes agree on the order of write operations to the same data item</td>
<td>Some sequential global order for writes to the same variables + programmed local order</td>
</tr>
<tr>
<td>Weak Consistency</td>
<td>Shared data can only be counted on to be consistent after a synchronization is done</td>
<td></td>
</tr>
<tr>
<td>Release Consistency</td>
<td>Shared data are made consistent by ‘acquire and release’ operations</td>
<td></td>
</tr>
<tr>
<td>Entry Consistency</td>
<td>Shared data are made consistent by per-variable ‘acquire and release’ operations</td>
<td></td>
</tr>
</tbody>
</table>