Shared Memory – Consistency of Shared Variables

The ideal picture of shared memory:

CPU0  CPU1  CPU2  CPU3

Read/Write

Shared Memory

The actual architecture of shared memory systems:

Symmetric Multi-Processor (SMP):

CPU0  CPU1  CPU2  CPU3

Read/Write

Local Cache  Local Cache  Local Cache  Local Cache

Shared Memory

R/W of Misses + Cache Invalidate

Distributed Shared Memory (DSM):

CPU0  CPU1  CPU2  CPU3

Local Memory Module  Local Memory Module  Local Memory Module  Local Memory Module

Network
The Million $$s Question: How/When Does One Process Read Other Process’s Writes?

Assumption: Initial value of shared variables is always 0.

Why is this a question? Because temporal order relations like “before/after” do not necessarily hold in a distributed system.
Why Memory Model?

Answers the question: “Which writes by a process are seen by which reads of the other processes?”
Memory Consistency Models

Example program:  

\[
\begin{align*}
\text{Pi:} & \quad R \ V; \ W \ V,7; \ R \ V; \ R \ V \\
\text{Pj:} & \quad R \ V; \ W \ V,13; \ R \ V; \ R \ V
\end{align*}
\]

A consistency/memory model is an “agreement” between the execution environment (H/W, OS, middleware) and the processes. Runtime guarantees to the application certain properties on the way values written to shared variables become visible to reads. This determines the memory model, what’s valid, what’s not.

Example execution:  

\[
\begin{align*}
\text{Pi:} & \quad R \ V,0; \ W \ V,7; \ R \ V,7; \ R \ V,13 \\
\text{Pj:} & \quad R \ V,0; \ W \ V,13; \ R \ V,13; \ R \ V,7
\end{align*}
\]

Order of writes to \( V \) as seen to Pi: (1) \( W \ V,7 \); (2) \( W \ V,13 \)  
Order of writes to \( V \) as seen to Pj: (1) \( W \ V,13 \); (2) \( W \ V,7 \)
Memory Model: Coherence

Coherence is the memory model in which (the runtime guarantees to the program that) writes performed by the processes for every specific variable are viewed by all processes in the same full order.

Example program: All valid executions under Coherence:

<table>
<thead>
<tr>
<th>Pi:</th>
<th>Pj:</th>
</tr>
</thead>
<tbody>
<tr>
<td>W V,7</td>
<td>W V,13</td>
</tr>
<tr>
<td>R V</td>
<td>R V</td>
</tr>
<tr>
<td>R V</td>
<td>R V</td>
</tr>
</tbody>
</table>

The Register Property: the view of a process consists of the values it "sees" in its reads, and the writes it performs. If a R V in P which is later than W V,x in P sees value different than x, then a later R V cannot see x.
Formal definition of Coherence

**Program Order:** The order in which instructions appear in each process. This is a *partial order* on all the instructions in the program.

**A serialization:** A *full order* on all the instructions (reads/writes) of all the processes, which is consistent with the program order.

**A legal serialization:** A serialization in which each *read* X returns the value written by the latest *write* X in the full order.

Let $P$ be a program; let $P_x$ be the “sub-program” of $P$ which contains all the *read* X/*write* X operations on X only.

**Coherence:** $P$ is said to be *coherent* if for every variable X there exists a legal serialization of $P_x$. (Note: a process cannot distinguish one such serialization from another for a given execution)
Examples

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read x,1</td>
<td>read y,1</td>
</tr>
<tr>
<td>write y,1</td>
<td>write x,1</td>
</tr>
</tbody>
</table>

Coherent. Serializations:
x: write x,1, read x,1
y: write y,1, read y,1

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read x,1</td>
<td>read x,2</td>
</tr>
<tr>
<td>write x,2</td>
<td>write x,1</td>
</tr>
</tbody>
</table>

Not Coherent.
Cycle of *dependencies*.
Cannot be serialized.

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>write x,1</td>
<td>read x,2</td>
</tr>
<tr>
<td>write x,2</td>
<td>read x,1</td>
</tr>
</tbody>
</table>

Not Coherent.
Cannot be serialized.
Sequential Consistency [Lamport 1979]

Sequential Consistency is the memory model in which all reads/writes performed by the processes are viewed by all processes in the same full order.

Coherent. Not Sequentially consistent.

Coherent. Not Sequentially consistent.
Strict (Strong) Memory Models

**Sequential Consistency:**
Given an execution, there exists an order of reads/writes which is consistent with all program orders.

**Coherence:**
For any variable $x$, there exists an order of read $x$/write $x$ consistent with all p.o.s.

\[ a=0, b=0 \]
\[ a=1, b=1 \]

Printed: 0,0 or 0,1 or 1,0
Printed: 1,1
Formal definition of Sequential Consistency

Let $P$ be a program.

**Sequential Consistency:** $P$ is said to be *sequentially consistent* if there exists a legal serialization of all *reads/writes* in $P$.

**Observation:** Every program which is sequentially consistent is also coherent.

**Conclusion:** Sequential Consistency has *stronger requirements* and we thus say that it is *stronger* than Coherence.

**In general:** A consistency model $A$ is said to be (strictly) stronger than $B$ if all executions which are valid under $A$ are also valid under $B$. 
The problem of strong consistency models

The runtime system should ensure the existence of legal serialization, and the same consistent view for all processes.

This requires lots of expensive coordination → degrades performance!

P1: Print(U)  Write V,1
P2: Print(V)  Write U,1

SC: Hardware cannot reorder locally in each thread for this will result in a possible printing 1,1. HW may reorder anyway and postpone writes, but then why reorder in the first place?
Coherence Forbids Reordering

Once thread sees an update – cannot “forget” it has seen it.

⇒ Cannot reorder two reads of the same memory location.
Coherence makes **reads** prevent common compiler optimizations

**Reads** can make a process see **writes** by another process. The read “kills” later reuse of local values.

p and q might point to same object

- p.x = 0
- p.x = 1
- a = p.x
- b = q.x
- c = p.x

assert(p==q \implies a \leq b \leq c)

Cannot put c = a
Release Consistency
[Gharachorloo et al. 1990, DASH]

• Introduces a special type of variables, called *synchronization variables* or *locks*.

• Locks cannot be read or written. They can be *acquired* and *released*, denoted `acquire(L)` and `release(L)` for a lock `L`.

• A process that acquired a lock `L` but has not released it, *holds* it.

• No more than one process can hold a lock `L`, while others wait.

Using release and acquire to define execution-flow synchronization primitives

- Let a set of processes release tokens by reaching the operation Release in their program order.
- Let another set (possibly with overlap) acquire those tokens by performing acquire operation, where acquire can proceed only when all tokens have already arrived from all releasing processes.
- 2-way synchronization = lock-unlock, 1 release, 1 acquire
- n-way synchronization = barrier, n releases, n acquires
- PARC’s synch = k-way synchronization
(Almost Formal) Definition of Release Consistency

<table>
<thead>
<tr>
<th>Assumption</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(A)</strong></td>
<td>Before a read or write access is allowed to perform, all preceding (program order) acquire accesses must be performed, and</td>
</tr>
<tr>
<td><strong>(B)</strong></td>
<td>Before a release access is allowed to perform, all preceding (program order) read or write accesses must be performed, and</td>
</tr>
<tr>
<td><strong>(C)</strong></td>
<td>acquire and release accesses are sequentially consistent.</td>
</tr>
</tbody>
</table>
Understanding RC

From this point in this execution all processes must see the value 1 in X

It is undefined what value is read here. It can be any value written by some process. Here it can be 0 or 1.

According to rule (B): 1 is read in the current execution. However, the programmer cannot be sure 1 will be read in all executions.

According to rules (A) and (C), the programmer knows that in all executions this read returns 1.
Acquire and Release

- release serves as a memory-synch operation, or a flush of the local modifications to all other processes.
- acquire and release are not only used for synchronization of execution, but also for synchronization of memory, i.e. for propagation of writes from/to other processes.
  - This allows to overlap the two expensive types of synchronization.
  - This turns out also simpler on the programmer from semantic point of view.
Acquire and Release (cont.)

- A release followed by an acquire of the same lock guarantees (the programmer) that all writes previous to the release will be seen by all reads following the acquire.
- The idea is to let the programmer decide which blocks of operations need be synchronized, and put them between matching pair of acquire-release operations.
- In the absence of release/acquire pairs, there is no assurance that modifications will ever propagate between processes.
Happened-Before relation induced by acquire/release
Data Races in RC

Release Consistency does not guarantee anything about ordered propagation of updates

Initially: grades = oldDatabase; updated = false;

• Thread T.A.
  grades = newDatabase;
  updated = true;
  while (updated == false);
  X:=grades.gradeOf(lecturersSon);

• Thread Lecturer

• If the modification of variable updated is passed to Lecturer while the modification of grades is not, then Lecturer looks at the old database!

• This is possible in Release Consistency, but not in Sequential Consistency.
Expressiveness of Release Consistency
[Gharachorloo et.al 1990]

**Theorem:** RC = SC for programs having no data-races.

Given a data-race-free program $P$, the set of valid executions for $P$ is the same on systems providing RC and those providing SC.

**Conclusion (OpenMP, Java, C++, etc):**
- System provide RC (performance)
- Programmer avoid data-races (program verification)
- ➔ Best of both worlds!
Lazy Release Consistency
[Keleher et al., Treadmarks 1992]*

• Postpone modifications until remote process “really” needs them
• More relaxed than RC

Formal Definition of Lazy Release Consistency

(A) Before a read or write access is allowed to perform with respect to any other process, all previous acquire accesses must be performed with respect to that other process, and

(B) Before a release access is allowed to perform with respect to any other process, all previous read or write accesses must be performed with respect to that other process, and

(C) acquire and release accesses are sequentially consistent.
Understanding the LRC Memory Model

• It is guaranteed that the acquirer of the same lock sees the modification that precede the release in program order.
Understanding the LRC Memory Model: Transitivity

- The process C sees the modification of $x$ by A.
Implementation of LRC

• Satisfying the happened-before relation between all operations is enough to satisfy LRC.
  – Maintenance and usage of such a detailed ordering would be expensive.

• Instead, the ordering is applied to process *intervals*.
  – Intervals are segments of time in the execution of a single process.
  – New interval begins each time a process executes a synchronization operation.
Intervals

P_1

1  2  3

rel(L_1)  acq(L_3)

P_2

1  2  3  4  5

acq(L_2)  acq(L_1)  rel(L_2)  rel(L_1)

P_3

1  2  3

rel(L_3)  acq(L_2)
Happened-before of Intervals

A happened before partial order is defined between intervals.

An interval $i_1$ precedes an interval $i_2$ according to **happened-before of intervals**, if all accesses in $i_1$ precede accesses in $i_2$ according to the **happened-before of accesses**.
**Vector Timestamps**

- An interval is said to be *performed* at a process if all interval’s accesses have been performed at that process.
- Each process $p$ has *vector timestamp* $V_p$ that tracks which intervals have been performed at that process.
  - A vector timestamp consists of a set of interval indices, one per process in the system.