

Consistency and Replication

Shared Memory

Shared memory architectures are the traditional domain for consistency problems. Types of architectures:

- shared memory symmetric multiprocessors:
 - all memory is shared and equidistant from processors
 - caching used for performance
- NUMA (non-uniform memory access architecture)
 - each machine has own memory but hardware allows accessing remote memory
 - remote addresses are no different than local accesses at assembly level
 - access to remote memory is much slower
 - no hardware caching occurs (but software may do caching)

- Distributed Shared Memory
 - software presents abstraction of shared memory
 - OS or language run-time manages shared memory
 - there are many different DSM granularities
 - page-based DSM: like regular virtual memory
 - shared-variable DSM and object-based DSM: managed by a language run-time system

Consistency Models

- Tannenbaum casts the consistency issue much broader than what was traditionally considered a shared memory problem.
- Instead, issue is cast in terms of consistency of replicated data.

Data-centric Consistency Models

- *Data store* physically distributed across multiple machines.
- Each process that can access data from data store is assumed to have a local copy available of *entire store*. Write operations are propagated to other copies.
- Consistency model : a contract between processes and the data store. If processes agree to obey certain rules, data store promises to work correctly.
- Normally, process that performs read on data item expects the value of last write. But difficult to define which write was last without global clock (and global clocks very difficult - as we will see later in semester.)

Consistency Models

A *Consistency Model* states that the data store will work correctly but only if the software obeys certain rules

The issue is how we can state rules that are not too restrictive but allow fast execution in most common cases

Conventions:

- $W(x)a$ means “a write to x with value a ”
- $R(y)b$ means “a read from y that returned value b ”

5 of 13

Strict Consistency

- Strictest consistency model
 - a read returns the most recently written value (changes are instantaneous)
 - expected semantics in uniprocessor system
 - this is what uniprocessors support:
`a = 1; a = 2; print(a);`
always produces “2”
 - to exercise our notation:
P1: $W(x)1$
P2: $R(x)0$ $R(x)1$
 - is this strictly consistent?
- relies on absolute global time
 - expects execution of commands is serialized centrally
 - effects of slow write may have not propagated to site of the read

6 of 13

Sequential Consistency

- Sequential consistency: results of any execution is same as if operations from different processes are executed in some sequential order. Operations of single process must appear in order specified by program
 - any valid interleaving of read and write operations is acceptable, but all processes *must see same interleaving of operations.*
- Example of sequentially consistent execution:
P1: $W(x)1$
P2: $R(x)0$ $R(x)1$
- Sequential consistency is inefficient: need consistency model that is weaker (hence cheaper)

7 of 13

Causal Consistency

- Causal consistency: operations that are causally related must be seen in causal order by all processes. Concurrent writes may be seen in different order on different machines.
- Concurrent writes may be seen in different order by different processes
 - causally related writes: P2's write of Y comes after P2's read of X that was written prior by P1.
- Concurrent operations - operations that are not causally related.
- Implementation must keep dependencies

8 of 13

• Examples (which one is causally consistent?)

```
P1: W(x) 1           W(x) 3
P2:   R(x) 1 W(x) 2
P3:   R(x) 1           R(x) 3 R(x) 2
P4:   R(x) 1           R(x) 2 R(x) 3
```

```
P1: W(x) 1
P2:   R(x) 1 W(x) 2
P3:           R(x) 2 R(x) 1
P4:           R(x) 1 R(x) 2
```

FIFO Consistency

- FIFO consistency is more relaxed than causal consistency: writes from same processor are received in order, but writes from distinct processors may be received in different orders by different processors
- No guarantees about order of writes except that two or more writes from single source must arrive in order.
- Key: every process can see different interleaving
- Easy to implement

```
P1: W(x) 1
P2:   R(x) 1 W(x) 2
P3:           R(x) 2 R(x) 1
P4:           R(x) 1 R(x) 2
```

Weak Consistency

- Weak consistency uses synchronization variables to propagate writes to and from a machine at appropriate points:
 - accesses to synchronization variables are sequentially consistent
 - no access to a synchronization variable is allowed until all previous writes have completed in all processors
 - no data access is allowed until all previous accesses to synchronization variables (by the same processor) have been performed
- That is:
 - accessing a synchronization variable “flushes the pipeline”
 - at a synchronization point, all processors have consistent versions of data

Weak Consistency Examples

- Which one is valid under weak consistency?
 - convention: S means access to synchronization variable

```
P1: W(x) 1 W(x) 2 S
P2:           R(x) 1 R(x) 2 S
P3:           R(x) 2 R(x) 1 S
```

```
P1: W(x) 1 W(x) 2 S
P2:           S R(x) 1
```

- Weak consistency means that programmer must manage synchronization explicitly

Release Consistency

- Release consistency is like weak consistency, but there are two operations “lock” and “unlock” for synchronization
 - doing a “lock” means that writes on other processors to protected variables will be known
 - doing an “unlock” means that writes to protected variables are exported
 - and will be seen by other machines when they do a “lock” (lazy release consistency) or immediately (eager release consistency)

- Example (valid or not?):

```
P1: L W(x)1 W(x)2 U
P2:           L R(x)2 U
P3:           R(x)1
```

- Variant: entry consistency: like lazy release consistency but data variables are explicitly associated with synchronization variables