Distributed Shared Memory and Data Consistency Models

Based on Tanenbaum/van Steen's "Distributed Systems", Ch. 6

section 6.2

also – will put online reference paper: Distributed Shared Memory, Concepts and Systems

Shared Memory

We have already seen some kinds of shared memory mechanisms

- shared memory symmetric multiprocessors:
  - all memory is shared and equidistant from processors
  - caching is used for performance

There are also other kinds of shared memory approaches

- NUMA (non-uniform memory access) machines
  - each machine has its own memory but the *hardware* allows accessing remote memory
  - remote addresses are no different than local accesses at the assembly level
  - access to remote memory is much slower
  - no hardware caching occurs (but the software may do caching)

- Distributed Shared Memory
  - objectives: minimize latency and keep coherent
  - software presents the abstraction of shared memory
  - either an OS or a language run-time manages the shared memory
  - there are many different DSM granularities
  - page-based DSM: like regular virtual memory (e.g., Clouds here at Tech!)
  - shared-variable DSM and object-based DSM: managed by a language run-time system

Think of how you would implement such a "virtual" shared memory among workstations

- at the OS level for unsuspecting processes
  - how can this be done efficiently?
- at a language runtime level (e.g., Java VM)
  - how can it be done without hardware support?

DSM vs. other shared memory mechanisms

- We have seen simple cache consistency mechanisms in two different settings:
  - SMPs (snooping caches, shared bus, write-update or write-invalidate protocols, write buffers, etc.)
  - distributed file systems (client or server-based protocols, leases, invalidation, delayed writes)
- Similar mechanisms apply to DSM, but they have to be more sophisticated
  - DSM will be used for synchronization and inter-process communication
  - DSM has to be very fast
  - comparable in speed to memory, not to a file system
  - caching is paramount
  - no central resources exist (like a bus we can snoop on or lock it cheaply)

Page-based DSM

- The issues in implementing DSM are similar to what we have seen so far, but some complications arise in the page-based case
  - which is the most common case for getting DSM running on machines with no shared memory support in hardware (e.g., workstations over ethernet)
- Pages are coarse grained—false sharing may occur
  - extra invalidations in a write-invalidate protocol
  - possibly overwriting other data in a write-update protocol
- Realistically, write-invalidate is the only option for page-based DSM
  - hard to do updates on every write: protection is at page granularity

Coherence Policy

- write update
  - allows multiple readers and writers
  - multicast message to update peer copies
  - processes have to agree on a total order for multicast writes for SC
- write invalidate
  - multiple readers, single writer
    - first invalidate peers
    - write to local cache
    - subsequent reads will get this new value

Granularity of Coherence

- page level coherence information
- false sharing a problem
  - solutions
    - smaller granularity
    - careful layout of data structures

Write-invalidate protocol

- Not particularly exciting—usually straightforward and exactly what you would expect
- Interesting point: the "owner" of a page can be changing (e.g., can be the last writer to the page)
- The owner is responsible for blocking "writes" until all outstanding copies have been invalidated
- How to find the owner?
  - directories
  - distributed directories (statically or dynamically)
  - hints (probable owner, owner links, and periodic broadcasts so that all processors know a recent owner)

Synchronization

- Strict synchronization (e.g., mutual exclusion) can be too costly in DSM
  - spinlocks are a disaster as they require transferring pages only to find out that the lock is still busy
- A centralized synchronization manager may exist

Sample Implementation Techniques

- DSM mapped to same VA on all nodes
- kernels at individual nodes responsible for page level protection
- page states: none, read-only, read-write
- two data structures:
  - owner(P): last writer of page P
  - copyset(P): current sharers for P
- home node (also called manager node)
  - where directory info for a page is kept
  - m nodes and N pages, distribute the work, i.e each node responsible for m/N pages
  - given a page, you can get the home node
- owner node
  - the node that has write permission for a page, at any given time

Consistency Models

- Both for DSM, but also for regular multiprocessors with real shared memory
  - i.e., either hardware or software could be enforcing these protocols
- (Tanenbaum 6.2)

Consistency Models

- A *Consistency Model* is a contract between the software and the memory
  - it states that the memory 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
- These models represent a more general view of sharing data than what we have seen so far!

Conventions we will use:

- *W(x)a* means "a *write* to *x* with value *a*"
- *R(y)b* means "a *read* from *y* that returned value *b*"
- "processor" used generically

Strict Consistency

- Strict consistency is the strictest model
  - a read returns the most recently written value (changes are instantaneous)
  - not even well-defined unless the execution of commands is serialized centrally
  - otherwise the effects of a slow write may have not propagated to the site of the read
  - 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?

Sequential Consistency

- Sequential consistency (*serializability*): the results are the same as if operations from different processors are interleaved, but operations of a single processor appear in the order specified by the program
  - really "in an order consistent with the program": the program does not strictly specify operation order. What if:

```
L1: a = 1;
if (a == 0) goto L1;
print(2);
goto L1;
```

- Example of sequentially consistent execution:

```
P1: W(x)1
P2:          R(x)0 R(x)1
```

- Sequential consistency is inefficient: we want to weaken the model further

Causal Consistency – by Tech people

- Causal consistency: writes that are potentially causally related must be seen by all processors in the same order. Concurrent writes may be seen in a different order on different machines
  - causally related writes: the write comes after a read that returned the value of the other write
- Examples (which one is causally consistent, if any?)

```
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
```

- Implementation needs to keep dependencies

Pipelined RAM (PRAM) or FIFO Consistency

- PRAM consistency is even more relaxed than causal consistency: writes from the same processor are received in order, but writes from distinct processors may be received in different orders by different processors

```
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
```

- Slight refinement:
  - **Processor consistency**: PRAM consistency plus writes to the same memory location are viewed everywhere in the same order

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
```

- Tanenbaum says the second is not weakly consistent. Do you agree? (What is the order of synchronizations? How do the rules prevent this execution?)
- Weak consistency means that the programmer has to manage synchronization explicitly

Release Consistency

- Release consistency is like weak consistency, but there are two operations “lock” and “unlock” for synchronization
  - (“acquire/release” are the conventional names)
  - 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 all data variables are explicitly associated with synchronization variables

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

a) Consistency models not using synchronization operations.

b) Models with synchronization operations.