Communication Models

- **message-passing**
  - example: UNIX sockets
  - disjoint address spaces: good between untrusted peers (client-server model)
  - communication is explicit: send/receive messages
  - synchronization is implicit: data available after receive
  - sender-based communication ("push")
  - programming is difficult: no pointers, data to communicate must be packed/unpacked in messages
  - RPC: partial alleviation: no pointers, no globals

- **shared memory**
  - example: threads
  - the same address space: good between trusted peers (parallel application)
  - communication is implicit: at memory accesses
  - synchronization is explicit: locks/mutex, conditional variables, monitors, barrier (global synchronization)
  - receiver-based communication ("pull")
  - programming is easy: pointers maintain their meaning because the address space is shared

Implementation

- **message-passing**
  - easy: a network with TCP/IP is enough
  - good scalability
  - inexpensive solution

- **shared memory**
  - difficult to build: sophisticated memory controllers
  - scalability is even more difficult to achieve
  - expensive solution

Dilemma: shared memory is simple to program but hardware is expensive to build; is this a cost-effective solution?
Shared Memory: Software Approach

- emulate shared memory in software on top of message-passing
  - simple programming
  - inexpensive hardware
- Kai Li’s idea (’86): SHARED VIRTUAL MEMORY (SVM)
  - implement shared memory in software as a virtual address space
  - emulate the functionality of a cache-coherent multiprocessor at page granularity using virtual memory
  - use a simple network of computers (with message-passing for communication)

Shared Virtual Memory (SVM)

- virtual address space

How is virtual memory used SVM

- virtual memory is used for page protection support
- if a page of the virtual address space is cached in the local memory then the page entry is valid (page is not protected)
- if a page of the virtual address space is not available in the local memory then the corresponding entry is invalid (page is protected for both read and write)
- an attempt to access a protected page causes a segmentation fault (page fault)
- the protocol handles the fault (by having set a fault handler)
- in the fault handler the protocol brings the faulting page from another node using message passing and unprotects the page
- everything is transparent to the programmer: looks like real shared memory

The coherence problem

- to speed up memory accesses we want to keep (cache) a virtual page in the local memory of the processors which access that page
- can be more than one -> page is replicated
- if we have more than one copy of the page, then when a write is performed we have a coherence problem:
  - how to make the write visible to other copies
  - how to combine writes performed on different copies of the same page (by different processors)
- we encountered the coherence problem before when we looked at caches and DMA transfer: how was it solved?
Simplest approach: no cache

- no caches -> an address location is stored in only one place: in memory
- processor writes to memory, NI reads from memory to send out messages
- memory accesses are slow because memory bus must be traversed for each access since there are no caches

With caches

- with caches: an address location can be stored in two places: in memory and in the cache -> replication
- CPU and NI must have a coherent view of memory
- when DMA out: data comes from cache if dirty there
- when DMA in: the cache block is invalidated

The multiprocessor case

- a memory bus, with several CPUs, one memory, no caches
- no coherence problem since there is no replication
- bus can become overloaded due to the memory traffic generated by processors
- memory accesses are slow

Cache-coherent multiprocessor

- adding caches introduces the coherence problem at writes
- snooping cache: caches observe the bus
- write through solution: memory is updated at each write
- other caches observe the write on the bus
- if they hold the word: either invalidate the entire cache block or update; if invalidate then the next access incurs a miss and the block is fetched from memory
Ownership-based caching

- write-through snooping cache still generates high traffic on the memory bus: absorb reads but not writes
- solution to absorb writes: make the cache write-back and give the processor exclusive access to the cache block at the first write (grant the ownership)
- further writes occurring on a owned cache block do not show up on the bus
- when another processor wants to read, the owner instead of memory provides the cache block and looses exclusivity
- when another processor wants to write ownership is transferred

Switched multiprocessors

- memory bus limits the scalability of bus-based multiprocessors
- alternative is to use a switched interconnect and memory physically distributed among nodes
- NUMA architecture (non-uniform memory access)
- CC-NUMA (cache-coherent NUMA), also called hardware distributed shared memory (DSM)
- the bus made coherence easy to maintain
  - broadcast medium allowed snooping
  - serialization medium allowed global ordering of events

Directory-based coherence

- directory keeps the current state and distribution of each cache block:
  - state of the cache block: invalid, shared, exclusive
  - copyset: processors which have valid copies of the cache block
- common approach: distributed directory
  - each node keeps a portion of the directory
  - each node is home for the cache blocks for which it keeps directory information
  - the home knows who is the current owner of the block
  - the same states but coherence protocol is complicated

State diagram

```
INVALID
       read fault
       invalidate

read fault (invalidate others)
write fault (ownership transfer)

read fault
write fault (invalidate others)

SHARE

read fault (invalidate others)

EXCLUSIVE
(at the owner)
```
DSM protocol

- on a read miss (INVALID -> SHARED)
  - send request to home
  - if block is SHARED send the block, update the copyset
  - if block is EXCLUSIVE, forward to owner, owner sends block to requester and home, home makes it SHARED
- on a write miss (INVALID/SHARED -> EXCLUSIVE)
  - send request to home
  - if block is SHARED, send invalidations to processors in copyset, wait for acks, make it EXCLUSIVE
  - if block is EXCLUSIVE, forward to owner, owner sends the ownership (and the block if requested), home updates owner

Back to SVM

- an SVM protocol is similar to a hardware directory-based DSM
  - pages instead of cache blocks
  - virtual memory instead of cache controller to maintain the page state
  - communication performed in software using message passing

False sharing

- in both hardware and software DSM coherence and communication is performed in coherence units: blocks or pages
- the protocol observes sharing at coherence unit granularity not at word/byte granularity as it occurs in the program
- consequently, the protocol cannot distinguish if two processors “really” share variables in the same coherence unit or if they access different variables which are collocated in the same coherence unit
- the latter is called “false sharing” and generates unnecessary communication
- occurs more often in SVM because pages are larger (4 KB page) than cache blocks in hardware DSM

Memory Consistency Problem

- has to do with the order of memory accesses
- in a bus-based multiprocessor, the memory bus is a serialization medium which guarantees a global ordering of events
- as a result, each processor sees memory accesses in the same order -> sequential consistency
- if we don’t have a bus (as in DSM and SVM systems) it is more difficult to implement sequential consistency
- but does the programmer really expect sequential consistency?
- the consistency model is a contract between the protocol and the programmer defining the restrictions on memory access ordering
**Sequential consistency (SC)**

- Assume: $a = b = 0$ initially
- $P_0$: $a = 1, \text{print } b$
- $P_1$: $b = 1, \text{print } a$

Any valid interleaving is acceptable as long as all processes see memory writes in the same order.

Legal outputs: sequential consistency: 01, 10, 11

- A read is not allowed to perform until previous writes are performed.
- A write is not allowed to perform until previous reads are performed.
- See the DSM protocol: “wait for acks” at writes is essential!

**What is wrong with SC**

- To implement SC in DSM/SVM we have to wait for writes to be performed (wait for acks) -> write latency is high.
- Read-write false sharing and write-write false sharing cause protocol to generate more communication than intended by the program (true sharing).
- In SVM the effect of false sharing can be dramatic: page thrashing.
- We would like to relax the consistency model to reduce the pressure on the protocol while still providing to the programmer a SC behavior if a set of rules is obeyed. A relaxed consistency model allows protocol optimizations.

**Processor consistency (PC)**

- Assume: $a = b = 0$ initially
- $P_0$: $a = 1, \text{print } b$
- $P_1$: $b = 1, \text{print } a$

Relaxation: writes done by the same processor must be received by all other processors in the order they were issued but writes from different processors may be seen in different orders by different processors.

Legal outputs: sequential consistency: 01, 10, 11, 00

- PC is interesting because most processors can support PC much easier than SC.
- Performance gap between PC and SC is significant: PC reduces write latency.

**What is wrong with PC**

- Even PC is too much for what programmers actually expect.
- Shared memory programs use synchronization to avoid race conditions and since they only access shared variables after synchronization (acquire a lock, for instance), they don’t care about the order in which writes are received.
- Relaxation: all writes performed before synchronization must be seen after synchronization = weak consistency.
- To implement weak consistency, program and protocol must distinguish ordinary variables from synchronization variables (in hardware DSM).
- In SVM synchronization is performed by explicit operations.
Release consistency (RC)

- a refinement of weak consistency
- distinguishes two kinds of synchronization operations:
  - acquire: get the lock, enter critical section
  - release: free the lock, exit critical section
- barrier is a global synchronization: process entering a barrier has to wait for all other processes to reach the barrier; it combines a release with an acquire:
  - arrival at the barrier is a release
  - departure from the barrier is an acquire

Release consistency (cont’d)

- main idea: if ordinary accesses are ordered with respect to synchronization accesses and if the program is race free, then the programmer cannot distinguish RC from SC
- before a release is allowed to perform, all previous writes done by the processors must be completed (rule 1)
- before an ordinary access is allowed to perform, all previous acquire done by the processor must be completed (rule 2)
- acquire, release operations must be sequential/processor consistent (rule 3)
- RC allows substantial protocol optimization: writes can be either pipelined (DSM) or buffered until release (SVM)

Under rules 1–3 RC == SC

Advantages of RC over SC

writes can be pipelined (DSM) or buffered (SVM)
reduces write latency, fewer messages (in DSM)
reduces unnecessary communication due to false sharing (for instance, at \(\text{print } c\) if page(c) == page(a))
What is wrong with (eager) RC

- Invalidations are propagated at release to all processes.
- Still exposes false sharing which can be eliminated by postponing the invalidations from release to acquire time.

Lazy Release Consistency (LRC)

- Invalidations are propagated lazily following the “happened before” relation and only to one processor at one time.
- LRC difficult to implement: propagate the transitive closure of invalidations with respect to “happened before” relation.

The “Happened Before” Relation

- “happened before” defines logical time: we don’t care about the absolute time when an event happened, only about the order between events.
- X --> Y means “X happened before Y
- Rules:
  1. If X and Y occur in the same processor, and X occurs before Y, then X --> Y.
  2. If M is a message, then send(M) --> receive(M).
  3. If X --> Y and Y --> Z, then X --> Z.

Logical Time Relationship:

- Given two events X and Y, either:
  1. X --> Y, or
  2. Y --> X, or
  3. neither: X and Y are concurrent (“X could not have caused Y, and Y could not have caused X”)
- --> relation defines a partial ordering.
- How to keep track of “happened before” relation in distributed systems (in particular in a LRC-based SVM)?
- Assign logical timestamps to each event.
Scalar Timestamps

- An unsigned integer
- Each processor has a “logical clock”
- Starts at zero, incremented on each local event
- Each message is timestamped with the sender’s logical clock
- On receive, receiver’s logical clock is set to \(1 + \max(\text{message timestamp}, \text{receiver's current logical clock})\)
- Captures “happened before” relationships, even more: \(\text{TS}(X)\) can be < \(\text{TX}(Y)\) but not \(X \rightarrow Y\)

Vector Timestamps

- Logical time represented as a vector of \(P\) entries (assuming \(P\) processors)
- \(\text{TS}(x)[i]\) incremented on every local event at \(l\)
- Message timestamped with the sender’s vector timestamp
- On receive, receiver sets \(J\)’s element of its vector to max of \(J\)’s element in message timestamp and \(J\)’s element of the current timestamp at receiver
- We say \(\text{TS}(X) < \text{TS}(Y)\), if \(\text{TS}(X)[i] < \text{TS}(Y)[i]\) for some \(i\) and all \(\text{TS}(X)[i] \leq \text{TS}(Y)[i]\)
- Captures “happened before” exactly: \(X \rightarrow Y\) if and only if \(\text{TS}(X) < \text{TS}(Y)\)

Using Vector Timestamp in L

- Logical time is defined by intervals: an interval is delimited by consecutive local synchronization events
- Each processor keeps its vector timestamps and records the invalidations received for each interval and processor (write notices)
- At acquire use timestamps to implement RC according to “happened before” relation:
  1. The acquirer sends its vector timestamp to the last releaser;
  2. The releaser sends back with the lock the write notices corresponding to intervals not seen at acquirer;
  3. The acquirer applies them and updates its vector timestamp

Lazy Release Consistency (L)

- The acquirer sends its vector timestamp to the last releaser;
- The releaser sends back with the lock the write notices corresponding to intervals not seen at acquirer;
- The acquirer applies them and updates its vector timestamp

\[
\begin{align*}
\text{P0} & \quad \text{acq}(L) \quad \text{a=1} \quad \text{rel}(L) \\
\text{P1} & \quad [1,0,0] \quad \text{acq}(L) \quad \text{b=1} \quad \text{rel}(L) \\
\text{P2} & \quad [1,1,0] \quad \text{acq}(L) \quad \text{print a print b} \quad \text{rel}(L)
\end{align*}
\]
Two key observations

1. Obs 1: no binding between locks and data is assumed: at acquire all the writes which “happened before” the corresponding release must be propagated (as invalidations) even if they were performed in a critical section protected by a different lock than the one which is acquired or even outside critical sections.
   - If we propagate only the writes performed in the critical sections protected by the same lock, we assume an explicit/implicit binding between data and locks -> different consistency model: entry/scope consistency.
2. Obs 2: RC and LRC reduces read-write false sharing but communication still occurs in the case of write-write false sharing.

Multiple writers

1. To alleviate write-write false sharing we must tolerate multiple writable copies between two synchronization events. The problem is how to merge the writes performed by multiple writers later on.
   - Assume a and b fall in the same page, i.e. page(a)==page(b).

Multiple writers using diffs

1. Before the first write, save a clean copy of the page: twin.
2. At the release, compare the page (dirty) with the twin (clean) and detect the differences: diff.
3. The faulting processor has to collect the diffs from all writers and apply them on its local copy to bring it up-to-date.

What is wrong with this scheme

1. In the worst case (all writers - all readers on the same page): O(n^2) messages and O(n) applications of each diff can lead to huge memory consumption for the protocol: diffs have to be kept around and eventually garbage collected.
2. Alternative scheme: Select a node as home of the page (usually the first writer) and propagate diffs eagerly to home.
Home-based LRC (HLRC)

1. In the worst case (all writers - all readers on the same page):
   - $O(n)$ messages and a single application of each diff (home)
   - diffs have short lifetime: used as vehicles to send writes to home, then immediately discarded
   - one remaining problem: must guarantee that writes sent as diffs are performed at home when page is fetched (no triangle inequality in network!):

```
+-----------------+-----------------+-----------------+
| P1              | P2              | P3              |
| acq(L2) b=1     | acq(L2)         | diff            |
| rel(L2)         | print a print b |                 |
+-----------------+-----------------+-----------------+
```

Guarantee Coherence in HLRC

1. Several solutions:
   1. Wait for acks from the home when you send the diff at release time -> makes release latency high because processors must wait for acks for all pages which were updated in the current interval
   2. Better solution: use vector timestamps to “version” the page
   3. Timestamp the diff with the logical clock when sending to home; home updates its page version after applying diff
   4. Use vector timestamps when requesting a page from home
   5. Home provides the page only when it has the required version (this guarantees memory coherence, i.e. page incorporates the necessary updates according to happened-before order)