Distributed Shared Memory

Marco Aiello, Ilche Georgievski

University of Groningen

DS, 2014/2015

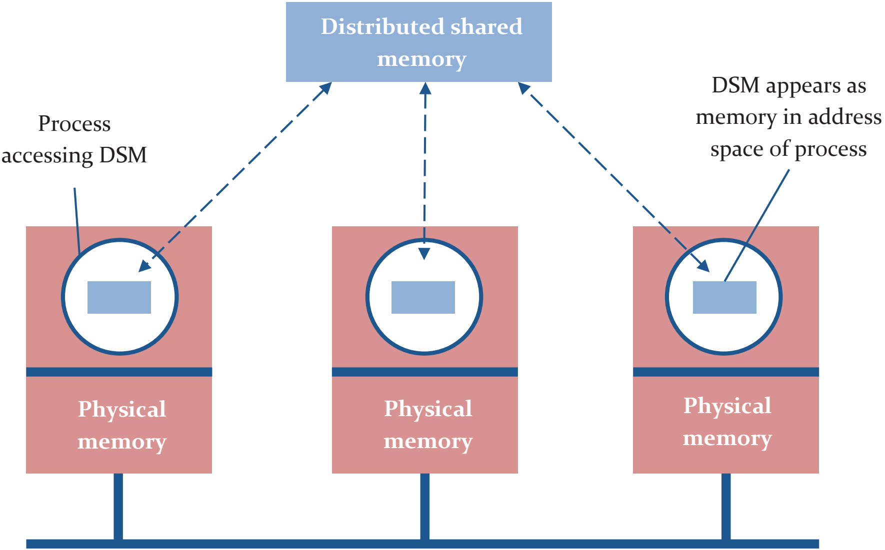
Outline

- Sharing data between processes that do not share physical memory
- When reading and updating, processes see DSM as an ordinary memory within their address space
- Processes on different computers observe the updates made by one another

Abstraction

It appears as processes access a single shared memory, but in fact the physical memory is distributed.

Abstration


Distributed shared memory
Process accessing DSM
DSM appears as memory in address space of process
Physical memory
Physical memory
Physical memory

- Best suitable when individual shared data items can be accessed directly (*e.g.*, parallel applications)

- Less appropriate when data is accessed by request (*e.g.*, client-server systems)

DSM vs. message passing

Property	DSM	Message passing
Marshalling	No	Yes
Address space	Single	Private
Data rep.	Uniform	Heterogeneous
Sync.	Normal construct for shared-memory programming	Message passing primitives
Process exec.	Non-overlapping	At the same time
Efficiency	No evidence against or in favour to any of the two communication mechanisms	
Cost visibility	Not necessarily explicit	Explicit

- No passing of messages
    - No marshalling of messages
    - Shields programmer from send/receive primitives

- But not absolutely
    - Sending messages for updates

- May be persistent - communication of processes living in different time intervals
- Programmer's point of view
  - ➠ Little programmer control
  - ➠ Programmers need to understand consistency models

Definition

A Distributed Shared Memory (DSM) is an abstraction that allows the physically separate memories to be addressed as one logically shared address space.

Central question

How to achieve a good performance that is retained as system scale to large number of computers?

Hardware approach

- Specialised hardware - *e.g.*, shared-memory multiprocessor architectures
  - Remote memory and cache modules
  - Network components

- Single address space

Page-based approach

- DSM as a region of virtual memory
- DSM has no particular structure
- Suited for homogeneous systems
- Usually implemented at user level
- Multiple addresses - memory of each processor organised in public and private addresses; the public part has a unique addressing scheme for all processes

Mether Writer

```
#include "world.h"
struct shared{int a,b;};

Program Writer:
    main()
    {
       struct shared *p;
       methersetup();                          /*Initialise the Mether run-time*/
       p = (struct shared *) METHERBASE; /*Overlay structure on METHER segment*/
       p->a = p->b = 0;                        /*Initialise fields to zero*/
       while(TRUE){                            /*Continuously update structure fields*/
           p->a = p->a + 1;
           p->b = p->b - 1;
       }
    }
```

Mether Reader

```
Program Reader:
     main()
     {
        struct shared *p;
        methersetup();
        p = (struct shared *) METHERBASE
        while(TRUE){                           /*Read the fields once every second*/
            printf("a= %d, b= %d\n", p->a, p->b);
            sleep(1);
        }
     }
```

Middleware approach

- No hardware or paging support - no use of existing shared-memory code
- Platform-neutral
- Sharing at user-level layer - higher-level abstractions of shared objects

Data structure

- Byte-oriented data
  - Data structure: flexible; a contiguous array of bytes
  - Direct sharing of memory elements seen as addressed collections of bytes
  - Two operations: *read* ($R(x)a$), *write* ($W(x)b$)
  - Example: $W(x)1$,$R(x)2$
- Object-oriented data
  - Data structure: a collection of language-level objects
  - Higher-level objects which are accessed only through invocations (*e.g.*, serialisation, easier for enforcing consistency)
- Immutable data
  - Data structure: a collection of immutable data items
  - Example: memory organised in tuples; processes share data by accessing the same tuple space; no direct access to tuples
  - Operations: *read*, *write*, *take*

Synchronisation model

Condition

```
a = b
```

Code

```
a := a + 1;
b := b + 1;
```

Synchronisation model

- Distributed synchronisation service
  - Synchronisation via locks and semaphores
  - Based on message passing
- Application-level synchronisation
- Synchronisation primitives make program easier to write and understand, but processor waiting for locks is wasting time

Consistency model

- Replication ➠ the issue of consistency
  - read: from local replica
  - update: to other replica managers

- Local replica manager: middleware + the kernel

Consistency model

- A memory consistency model specifies the consistency guarantees that DSM gives about the read values
- Models:
  - *weak*: writes propagate to replicas on demand
  - *strong*: a write propagates to all replicas before any read

Example

Process 1

```
br := b;
ar := a;
if(ar ≥ br) then
  print("OK");
```

Process 2

```
a := a + 1;
b := b + 1;
```

Atomic consistency

For any execution, there exists an interleaving of series of operations s.t.

1. the interleaving sequence of operations meets the specification of as a single correct copy of the objects
   ➠ if $R(x)a$ occurs in the sequence, then either the last write operation that occurs before $R(x)a$ in the interleaved sequence is $W(x)a$, or no write operation occurs before $R(x)a$, and a is the initial value of x
2. the order of operations is consistent with the real times of the operation execution

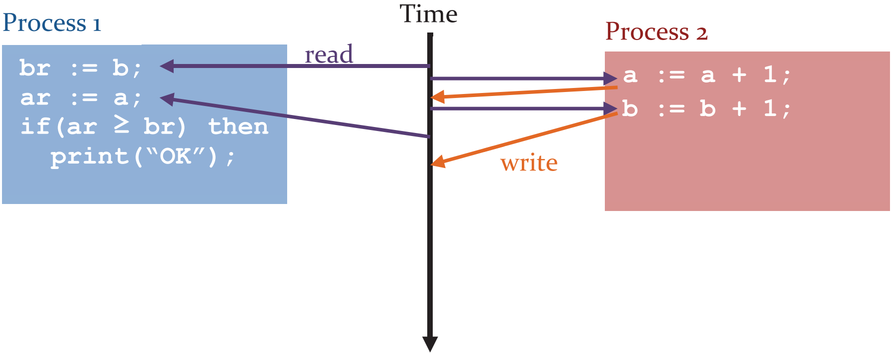
Sequential consistency

For any execution, there exists an interleaving of series of operations s.t.

1. same as the previous one
2. the order of operations is consistent with the program order in which each client has executed them

* The strongest model used in practice, a costly model to implement

Example (interleaving)



Coherence

- A weaker model with well-defined properties
- Every process agrees on the order of *write* operations to the same location
- Processes do not necessarily agree on the ordering of *write* operations to different locations
- $\approx$ a sequential consistency on a location-based basis
    - Saving: accesses to two different pages are independent and do not delay each other

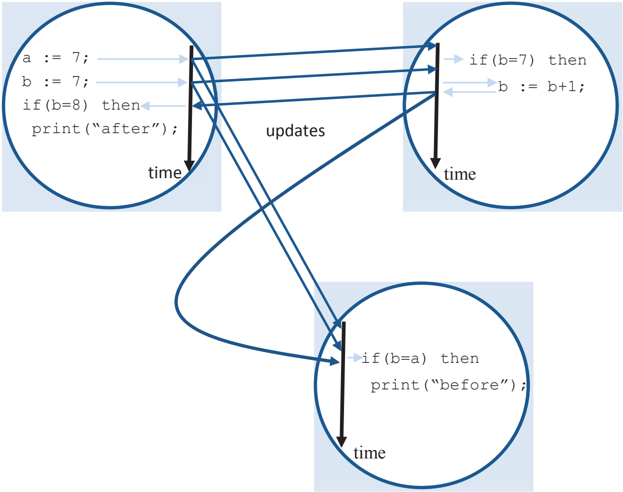
Weak consistency

- Avoids the costs of sequential consistency, but retains the effects of sequential consistency
- Synchronisation of operations to relax memory consistency
- *e.g.*, do not propagate writes while code execution in a synchronised fragment

Write-update

- Local updates are immediately multicast to all other replica managers possessing copies of the data item
- Multiple-reader/multiple-writer sharing
- Cheap reads
- The consistency model depends on the ordering of multicast (*e.g.*, totally ordered multicast for sequential consistency)

Example


a := 7;
b := 7;
if(b=8) then
print("after");
time
updates
if(b=7) then
b := b+1;
time
if(b=a) then
print("before");
time

Write-invalidate

- Multiple-reader/single-writer sharing
- Read-only mode: a data item can be copied indefinitely to other processes
- Write mode:
  - multicast to other processes to invalidate their copy and lock the item
  - access to the item: first come, first serve
- Achieves sequential consistency
- Expensive invalidation of read-only copies
  - trade-off: sufficiently high read/write ratio

Granularity

- The atomic size of memory to be shared (what should be the unit of sharing?)
- Fine-grained: less contentions and conflict, but network overhead
- Coarse-grained: more contentions (false sharing, useless invalidation) but better network performance
    - * In practice: the choice of the unit of sharing has to be made based on the physical page size available

Trashing

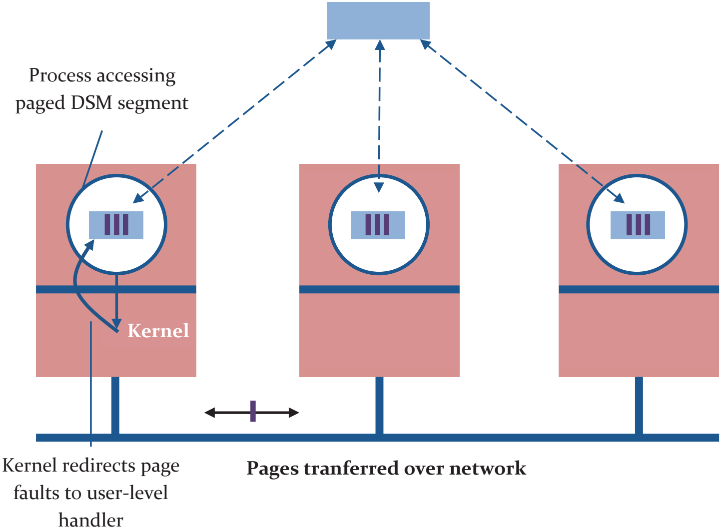
- Write-invalidate: the system spends most of its resources invalidating and updating memory fragments, without carrying on any useful computation
- Several processes compete for the same data item
- Several processes compete for falsely shared data items

Case study: IVY

About

- One of the first designs for a DSM runtime system (Yale University)
- Hardware: cheap PCs and a LAN (off-the-shelf hardware)
- Two classes of memory: private and shared
- Same address space for all processes
- Page-based implementation
- Sequential consistency
- Paging is transparent
  - Memory management unit
  - Permissions: *none*, *read-only*, *read-write*

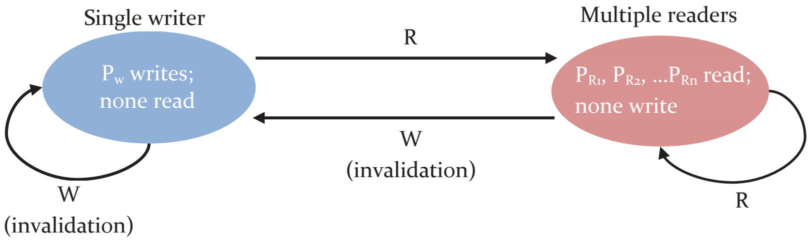
System model



Write invalidation

- Page protection to enforce consistent data sharing
- Multiple reader/single writer semantics
    - $owner(p)$ - an owner (the single writer or one of the readers) of the most up-to-date page p
    - $copyset(p)$ - a set of processes that have a copy of page p

State transitions



R: read fault occurs

W: write fault occurs

Write page fault

- The page fault handler acquires the page from the current holder
- Procedure:
  - the page is transferred to P_w (if it does not have an up-to-date read-only copy)
  - all other copies are invalidated (permissions are set to *no access* to all members of $copyset(p)$)
  - $copyset(p) := \{P_w\}$
  - $owner(p) := P_w$
  - the page with read-write permissions is placed at the appropriate location in its address space and the faulting instruction is restarted

Read page fault

- The page fault handler acquires the page from the current holder
- Procedure:
  - the page is copied from $owner(p)$ to P_R
  - if the current owner is a single writer, it remains the owner and its access permission is set to read-only access.
  - $copyset(p) := copyset(p) \cup \{P_R\}$
  - $owner(p) := P_w$
  - the page with read-write permissions is placed at the appropriate location in its address space and the faulting instruction is restarted

Invalidation protocols

- How to locate $owner(p)$ for a given page p?
- Where to store $copyset(p)$?

Centralised management algorithm

- A single manager stores the location of $owner(p)$, which stores $copyset(p)$
- A manger can be any process (running the application or any other)

Fixed distributed management algorithm

- Multiple managers, each acts as a central manager
- Pages are divided statically between them
- Some managers might incur more load than others

Multicast-based distributed management algorithm

- A faulting process multicasts its page request, the process owning the page replies
- Important: if two clients request the same page at more or less the same time
  - Totally ordered multicast
  - Unordered multicast where each page has associated a vector timestamp
- Disadvantage: processes that are not owners of a page are interrupted by irrelevant messages

Dynamic distributed management algorithm

- Page ownerships is transferred between processes
- Every process keeps, for every page, keeps a hint as to the page's current owner
- The owner is located by following the chains of hints
- To avoid inefficiency, hints are updated with the most recent values available

Faults

In all three implementations, the double fault problem is inherent. Successive read and write accesses to a page on a single node cause the page to be transferred twice. The authors provide a scheme to eliminate this problem using sequence numbers for every shared page. IVY's synchronisation primitives, which are needed to serialise concurrent accesses to shared memory locations, are called eventcounts. These eventcounts are atomic operations on shared counters which are implemented through the system's shared memory semantics.

Summary

- Parallel processing and data sharing
- Difficult to achieve efficient implementation
- Performance varies with types of applications
- Data structure: a series of bytes, a collection of objects, a collection of immutable data
- Application-level synchronisation
- Sequential consistency as the most common model in practice

Summary

- Update options
  - Write-update: updates propagated to all copies as data items are updated (hardware and software implementations)
  - Write-invalidation: prevent stale data being read by invalidating copies as data items are updated (page-based implementations)
- Granularity
  - Contention between processes that falsely share data items
  - Cost per byte of transferring updates

Is DSM a successful idea?

- Spreading computation across machines?
  - Yes, Google as an example
- Coherent access to shared memory?
  - Yes, multiprocessor PCs use IVY-like protocols for cache coherence between PCs
- DSM as a model for programming workstation cluster?
  - Hard to say: little evidence of adoption, limited control over communication

References

- G. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems: Concepts and Design, 4th Edition, Chapter 18, *Addison Wesley/Pearson Education*, 2005
- D. Patterson, J. Hennessey, Computer Organization and Design, 4th Edition, *Elsevier*, 2011
- B. Karp, Distributed Shared Memory: Ivy, Lecture Slides, University College London, 2009
- If you want to know more:
  - G. Hermannsson, L. D. Wittie, Optimistic Synchronization in Distributed Shared Memory, In *the Proceedings of the International Conference on Distributed Computing Systems*, pp. 345-354, 1996
  - K. L. Johnson, M. F. Kaashoek, D. A. Wallach, CRL: High-Performance All-Software Distributed Shared Memory, In *the Proceedings of the 15th ACM Symposium on Operating Systems Principles*, ACM Operating Systems Review, SIGOPS, vol. 29(5):213-226, 1995