Distributed Shared Memory

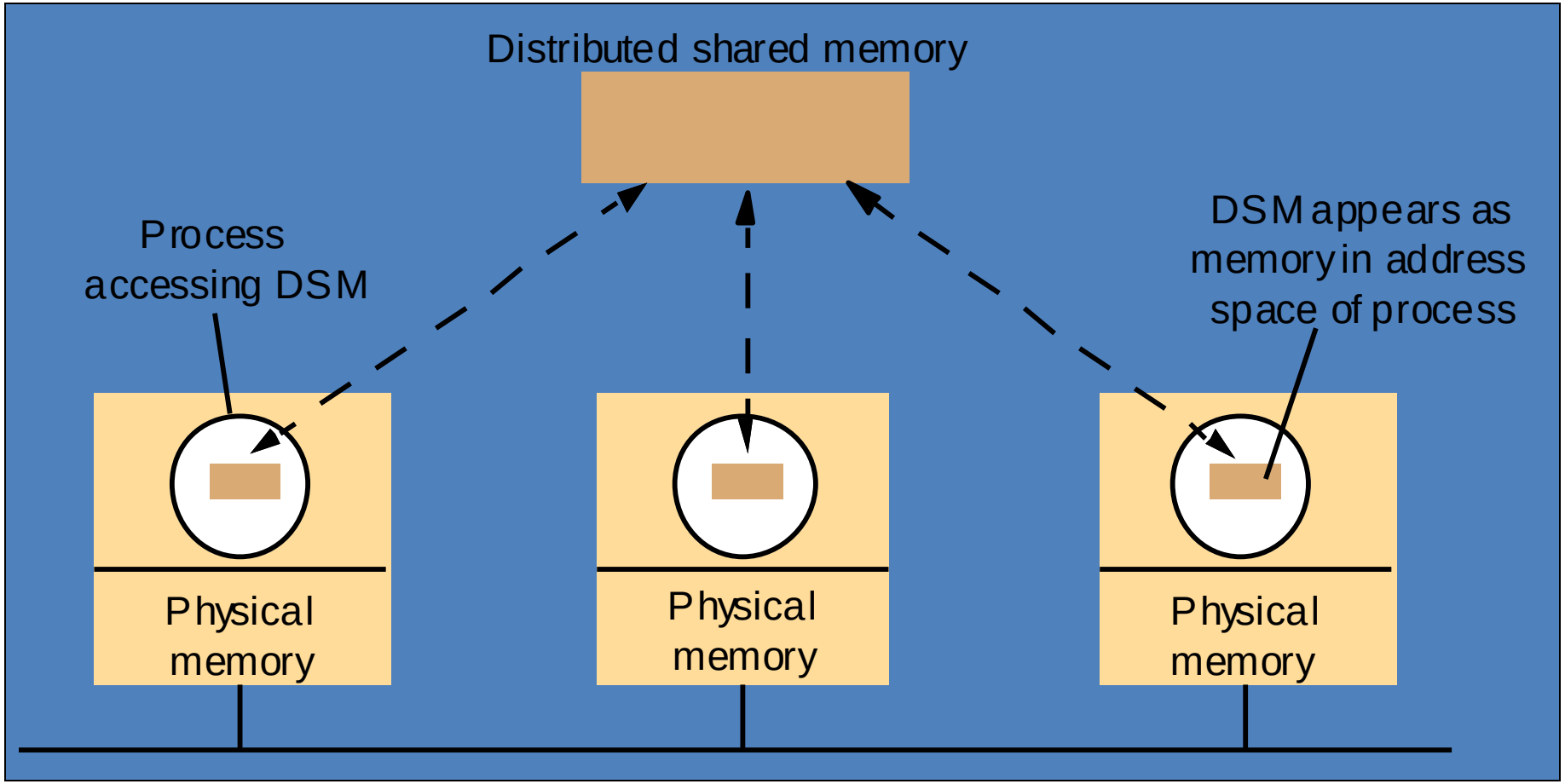
Distributed Shared Memory

- Distributed Shared Memory (DSM) allows programs running on separate computers to share data without the programmer having to deal with sending messages.
- Instead, underlying technology will send the messages to keep the DSM consistent (or relatively consistent) between computers.
- DSM allows programs that used to operate on the same computer to be easily adapted to operate on separate computers.

Introduction

- Programs access what appears to them to be normal memory.
- Hence, programs that use DSM are usually shorter and easier to understand than programs that use message passing.
- However, DSM is not suitable for all situations. Client-server systems are generally less suited for DSM, but a server may be used to assist in providing DSM functionality for data shared between clients.

The distributed shared memory abstraction



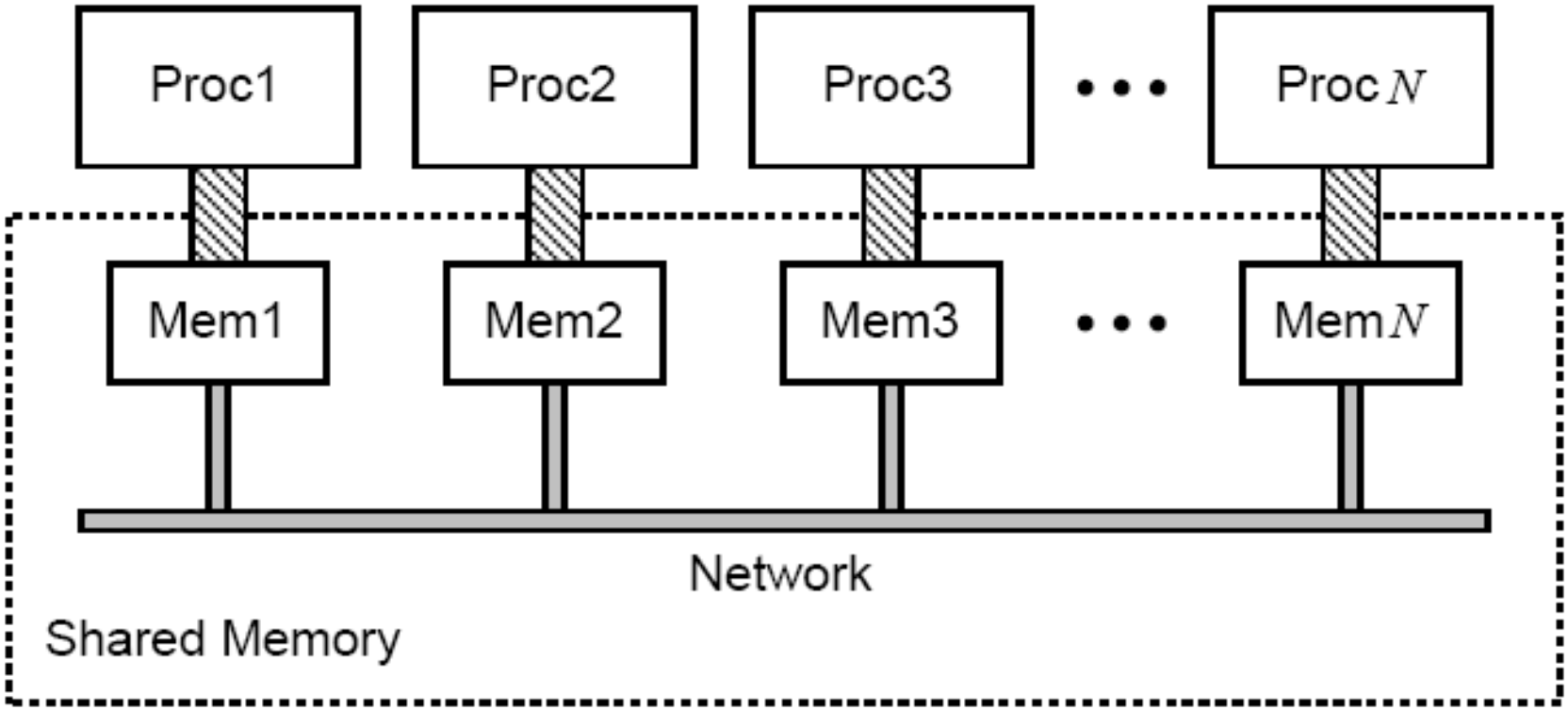
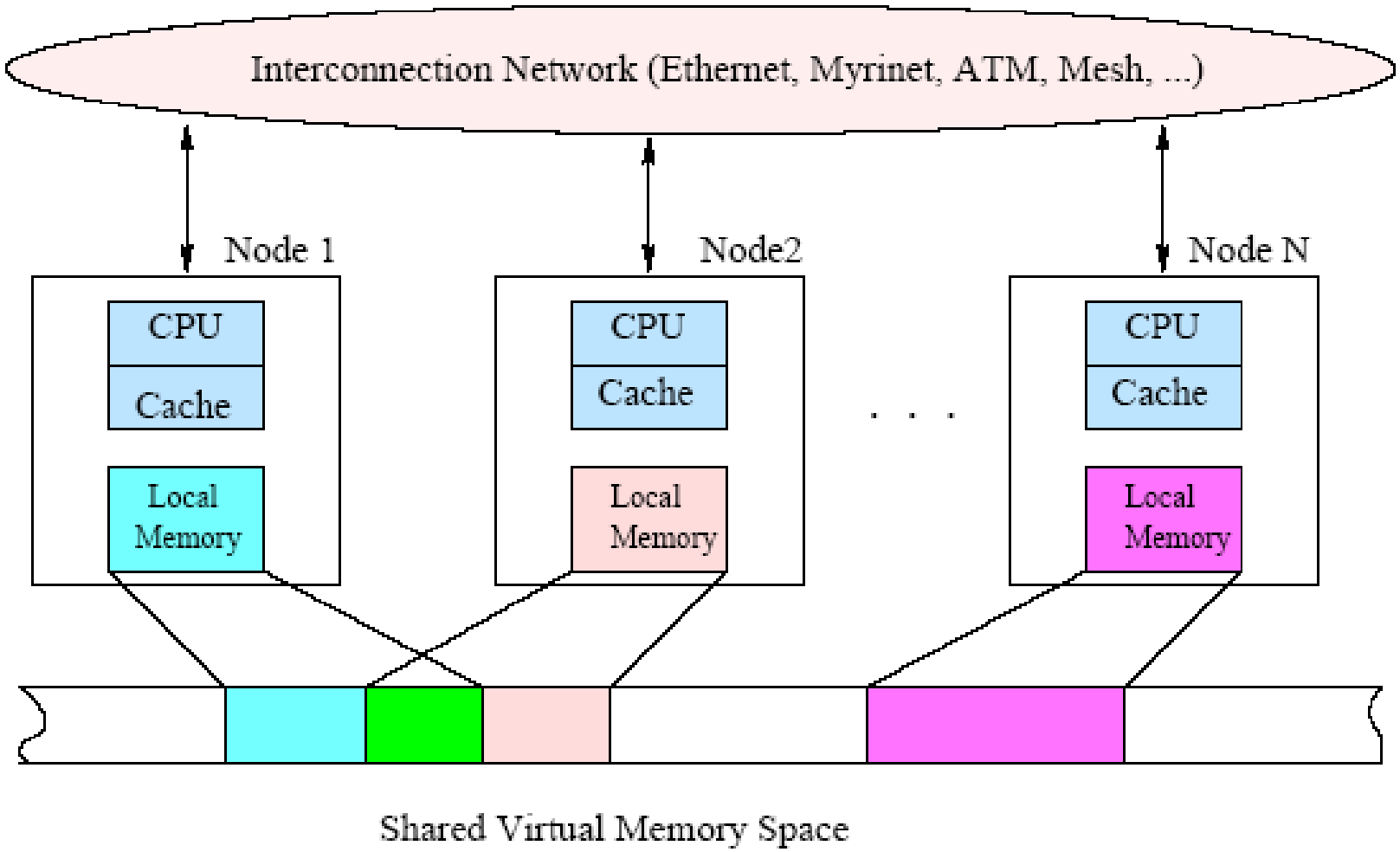


Figure 1 Distributed Shared Memory


Interconnection Network (Ethernet, Myrinet, ATM, Mesh, ...)
Node 1
Node2
Node N
CPU
Cache
Local Memory
CPU
Cache
Local Memory
CPU
Cache
Local Memory
Shared Virtual Memory Space

DSM History

- Memory mapped files started in the MULTICS operating system in the 1960s.
- One of the first DSM implementations was Apollo. One of the first system to use Apollo was Integrated shared Virtual memory at Yale (IVY).
- DSM developed in parallel with shared-memory multiprocessors.

DSM vs. Message Passing

DSM	Message Passing
Variables are shared directly	Variables have to be marshalled first
Processes can cause error to one another by altering data	Processes are protected from one another by having appropriate address space
Processes may execute with non-overlapping lifetimes	Processes may execute at the same time
Invisibility of communication cost	Cost of communication is exist
Synchronization in processes using locks and semaphores	Synchronization using Message passing primitives

DSM Implementations

- Hardware: Mainly used by shared-memory multiprocessors. The hardware resolves LOAD and STORE commands by communicating with remote memory as well as local memory.
  - E.g. NUMA Multiprocessor Architecture

- Paged virtual memory: Pages of virtual memory get the same set of addresses for each program in the DSM system. This only works for computers with common data and paging formats. This implementation does not put extra structure requirements on the program since it is just a series of bytes.

DSM Implementations (continued)

- Middleware: DSM is provided by some languages and middleware without hardware or paging support. For this implementation, the programming language, underlying system libraries, or middleware send the messages to keep the data synchronized between programs so that the programmer does not have to.

Design and Implementation Issues

- Structure of data held in DSM.
- Synchronization model used to access DSM
- DSM Consistency Model: consistency of data values
- Update options for communicating written values accessed from different computers.
- Granularity of sharing DSM
- Problem of Thrashing

Design approaches

- Byte-oriented: This is implemented as a contiguous series of bytes. The language and programs determine the data structures.

  R(x) a: reads value ‘a’ from location x

  W(x) a: stores value ‘a’ at location x
- Object-oriented: Language-level objects are used in this implementation. The memory is only accessed through class routines and therefore, OO semantics can be used when implementing this system.
- Immutable data: Data is represented as a group of many tuples. Data can only be accessed through read, take, and write routines.
  - Take: returns the tuple but removes the tuple from space.

Memory Consistency Models

- What is Memory consistency?
  - DSM replicates the contents of shared memory by caching it at separate computers.
  - Two operations:
    - W(*var*)*value*: write value to shared variable ‘var’.
    - R(*var*)*value*: read shared variable ‘var’ obtaining the value.

P1: w(x)1 R(x)1

----------→ time

P1: w(x)1

P2: R(x)?

----------→ time

Memory Consistency Models

- Determine when data updates are propagated and what level of inconsistency is acceptable.
- A consistency model is essentially a contract between the software and the memory.
- If the software agrees to obey certain rules, the memory promises to work correctly.
- If the software violates these rules, correctness of memory operation is no longer guaranteed.

Memory Consistency Models

- Strict consistency
- Sequential consistency (SC)
- Release consistency (RC)

Memory Consistency Models

- Strict consistency
  - Definition: any read to memory location x returns the value stored by the most recent write operation to x.
  - Also called atomic consistency or Linearizability

P1: w(x)1
P2: R(x)1
----------→ time

Memory Consistency Models

- Sequential consistency (SC)
  - Definition: the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.
  - Any valid interleaving is acceptable behavior, but all processes must see the same sequence of memory reference.

Memory Consistency Models

- **Sequential consistency** (SC)

processor 1: <-- A1 run --> <-- B1 run --> <-- C1 run -->
processor 2: <-- A2 run --> <-- B2 run -->
Time -->

Memory Consistency Models

- Release Consistency (RC)
  - Two operations:
    - Acquire Access: request entry to CS.
    - Release Access: completed operations.
  - Two types of access
    - Ordinary access: read and write
    - Synchronization access: acquire lock, release lock and barrier
  - Rules:
    - Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.
    - Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.

- Release Consistency (RC)

  Rule1: …Acq(L1)…Rel(L1) … **Acq(L5)** …**Acq(L3)**…**R(x)**

  Rule2: Acq(L2) **w(x) R(y) R(z) Rel(L2)**

  Example:

  P1: **Acq(L)** w(x)1 w(x)2 **Rel(L)**

  P2: **Acq(L)** R(x)2 **Rel(L)**

  P3: R(x)?

  Two types of Release Consistency:
  - Eager
  - Lazy

Weak Consistency

- It is useful when accesses are divided into synchronizing and non-synchronizing accesses.
  - Following properties:
    - Access to synchronization variables are sequentially consistent.
    - No access to synchronization variable is allowed to be performed until all previous writes have completed elsewhere.
    - No data access (Read/Write) is allowed to be performed until all previous accesses to synchronization variables have been performed.
- Therefore, there can be no access to synchronization variable if there are pending write operations. And there can not be any new read/write operation started if system is performing any synchronization operation.

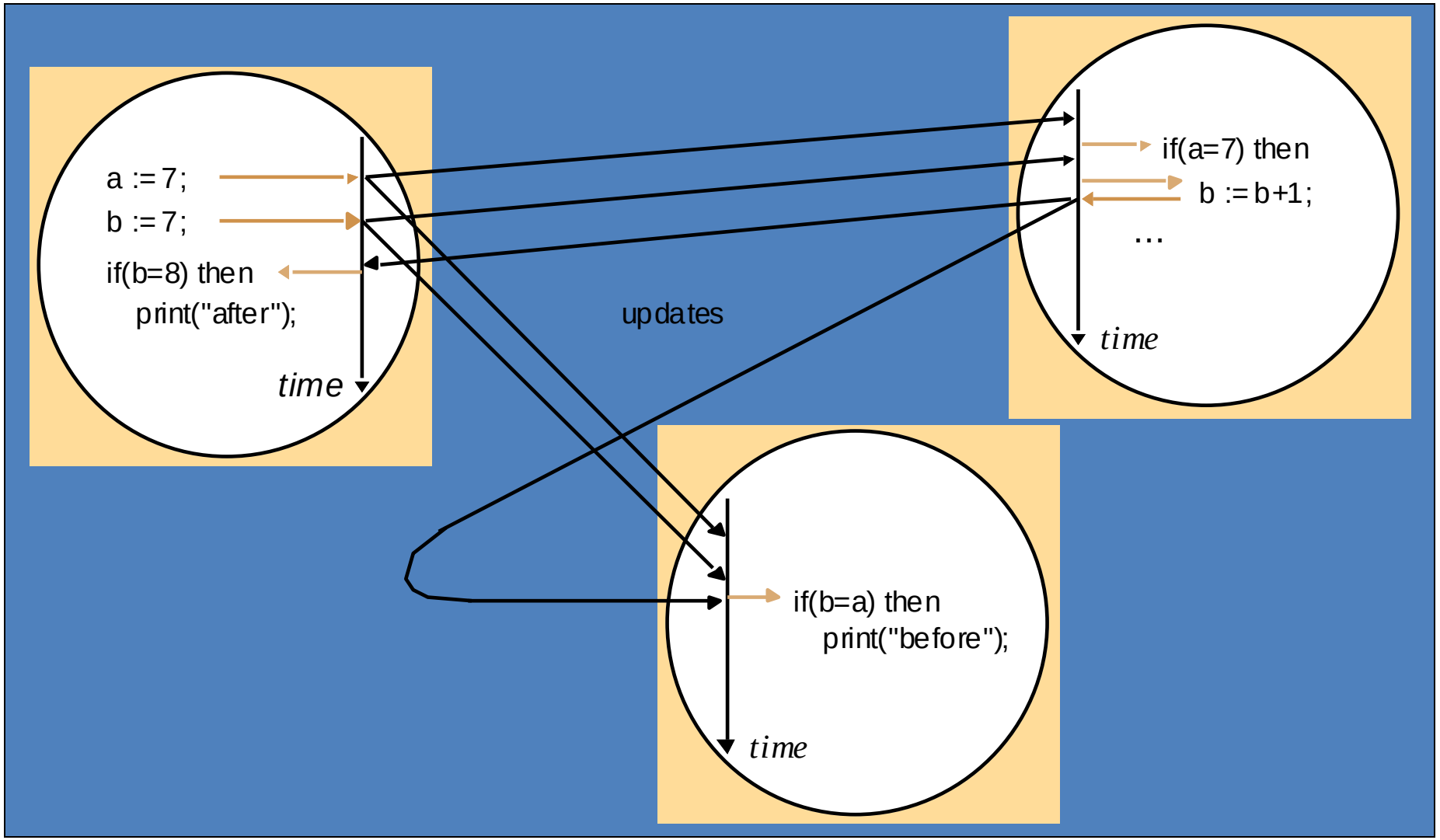
Cache Coherency

- Consistency of data stored in local cached of a shared resource.
- Coherence has significantly weaker consistency. It ensures writes to individual memory locations are done in the proper order, but writes to separate locations can be done in improper order.

Update options

- Write-update: Each update is multicast to all programs. Reads are performed on local copies of the data without communication.
  - Implemented for multiple-reader/multiple-writer.
  - Multicast ordering property.
- Write-invalidate: A message is multicast to each program invalidating their copy of the data before the data is updated. Other programs can request the updated data.

DSM using write-update

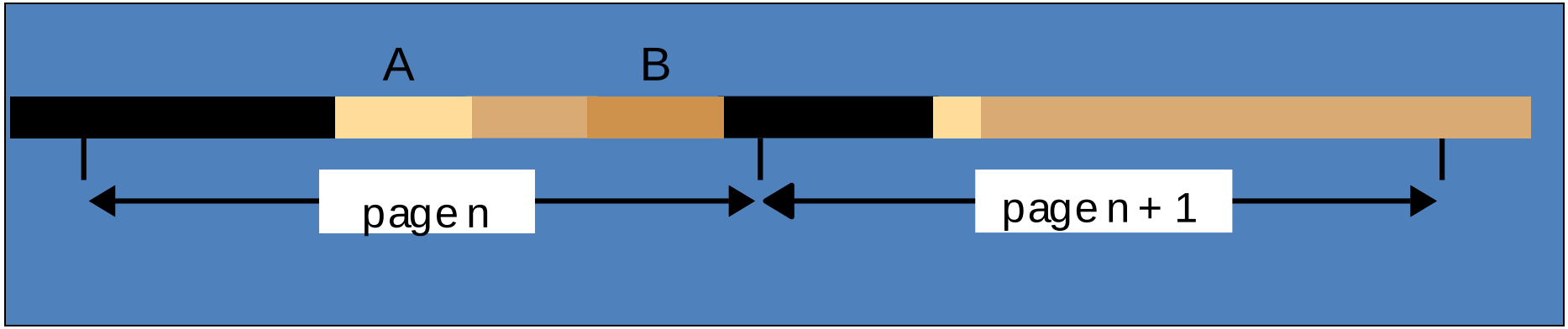


Granularity

- What should be the unit of sharing in DSM implementation?
- Granularity is the amount of data sent with each update.
- Page-based implementation:
  - If granularity is too small and a large amount of contiguous data is updated, the overhead of sending many small messages leads to less efficiency.
  - If granularity is too large, a whole page (or more) would be sent for an update to a single byte, thus reducing efficiency.

Granularity

- False Sharing: Two or more processes share parts of a page, but only one in fact accesses each part.



Effects of False Sharing:

- Write invalidate: unnecessary invalidations
- Write Update: Several writers falsely share data items, cause of overwritten with older version.

Thrashing

- Thrashing occurs when network resources are exhausted, and more time is spent invalidating data and sending updates than is used doing actual work.
  - E.g. One process repeatedly reads a data item that another is regularly updating, then item will be constantly transferred from writer to reader.
- Based on system specifics, one should choose write-update or write-invalidate to avoid thrashing.