Distributed Operating Systems (CS30009)

Overview of Distributed Shared Memory

Dr. Jaydeep Das

Contents

- Architecture
- Protocols
- Consistency Models (Covered in Data-Centric Consistency Models in unit 4)
- Page – Based Distributed Shared Memory
- Shared-Variable Distributed Shared Memory
- Object-Based Distributed Shared Memory

Introduction

- **Multiprocessor**
  - Two or more CPUs share a common main memory.
  - Any process on any processor can read/write any word in the shared memory by moving data to/from the desired location
- **Multicomputer**
  - Each CPU has its own private memory. Nothing is shared
- **Design Issues in Multiprocessor**
  - Hardware: Many processors access same memory simultaneously is difficult to design a machine
  - Bus-based Multiprocessor: Cannot be used more than a few dozen processors because bus tends to a bottleneck
  - Switched-based Multiprocessor: Can be made in large scale, but it is expensive, slow, complex and maintenance problem.

Introduction (contd…)

- From hardware designer's perspective, multi-computers are preferable than multiprocessor.
- **Software Issues in Multicomputer**
  - Many techniques are available for programming multiprocessor for communication, synchronization.
  - In multicomputer, communication is done through message passing making input/output the central abstraction
  - Message passing arises many complications like flow control, lost messages, buffering, and blocking
- Multiprocessor is harder to build but easier to program
- Multicomputer is easier to build but harder to program

Architecture

Introduction

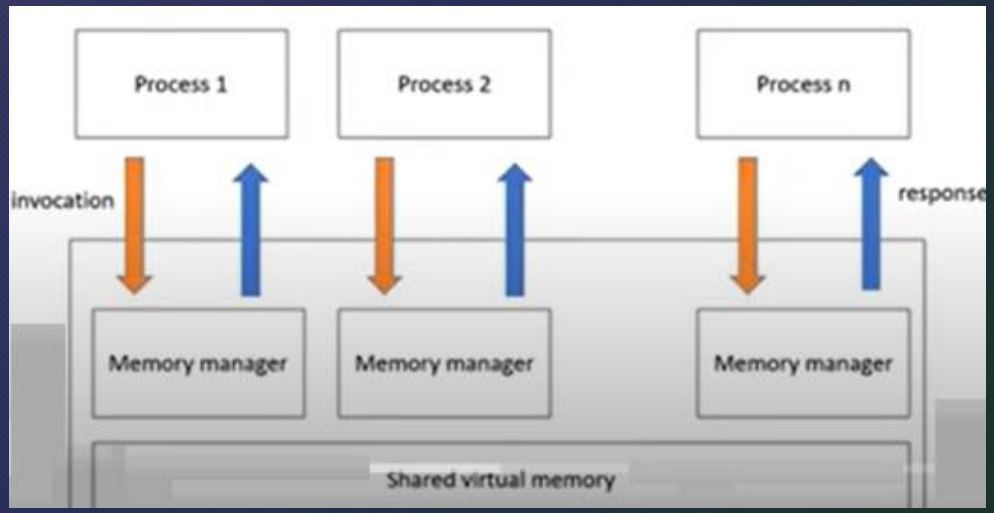
- On-Chip Memory
- Bus-Based Multiprocessors
- Ring-Based Multiprocessors
- Switched Multiprocessors
- Directories
- Caching

Introduction

- Distributed Shared Memory (DSM) is a mechanism that manages across multiple nodes and makes inter-process communication transparent to end-users
- The applications will think that they are running on shared memory
- DSM is a mechanism of allowing user processes to access shared data without using inter-process communication
- In DSM every node has its own memory and provides memory read and write services and it provides consistency protocols
- DSM implements the shared memory model in distributed systems but it doesn’t have physical shared memory

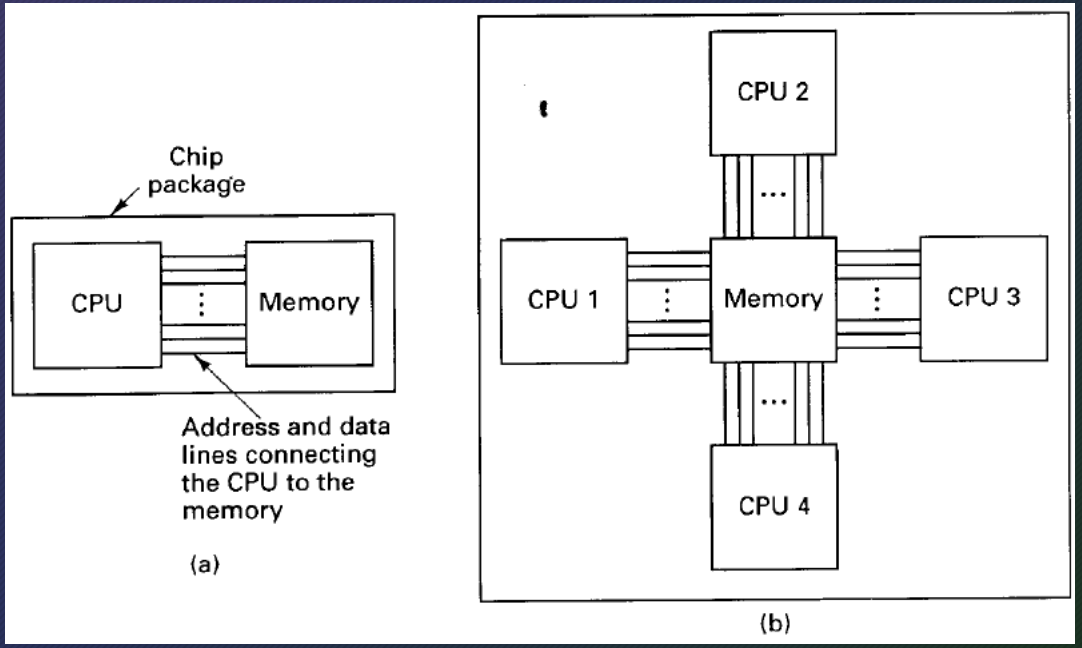
Distributed Shared Memory

- All the nodes share the virtual address space provided by the shared memory model. The data moves between the main memories of different nodes.



Types of DSM

1. On-chip Memory
   - The data is present in the CPU portion of the chip.
   - Memory is directly connected to address lines
   - On-Chip Memory DSM is expensive and complex



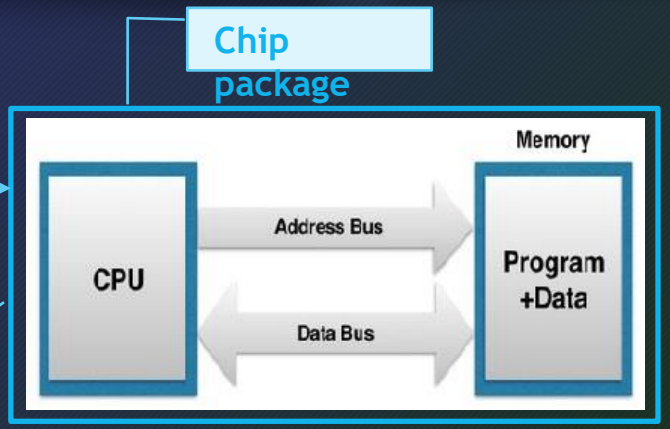
(a) A single-chip computer (b) A hypothetical shared-memory multiprocessor

1. On-Chip Memory

- On-Chip Memory

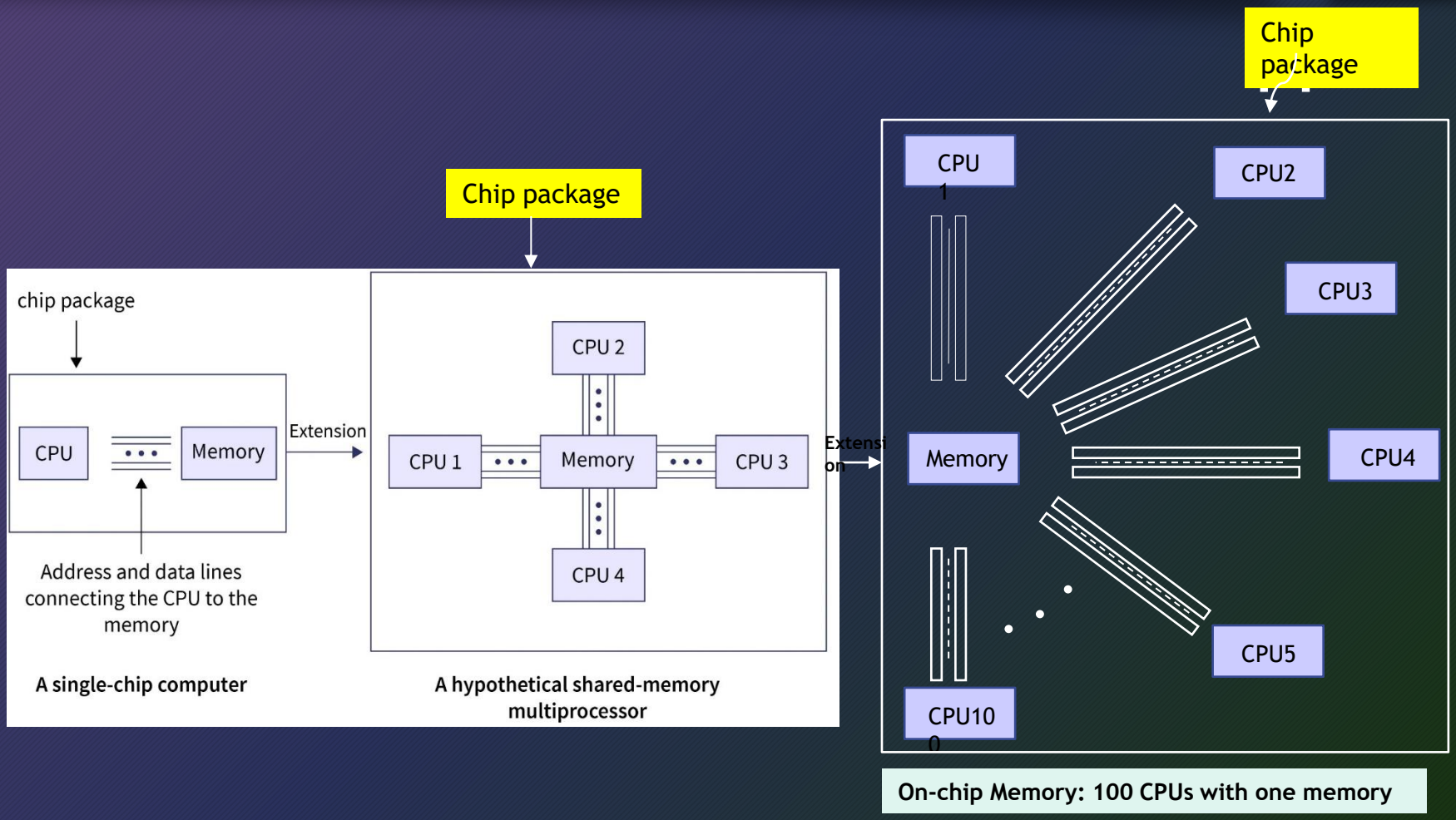
Self-contained chips containing a CPU and all the memory also exist. Such chips are produced by the millions, and are widely used in cars , appliances and even toys. The CPU portion of the chip has address and data lines that directly connect to the memory portion.

One could imagine a simple extension of this chip to have multiple CPU's directly sharing the same memory. Constructing such a chip would be complicated, expensive and highly unusual.



The CPU portion of the chip has address and data lines that directly connect to the memory portion. This types of chips are used in cars, toys, appliances & electronic gadgets.

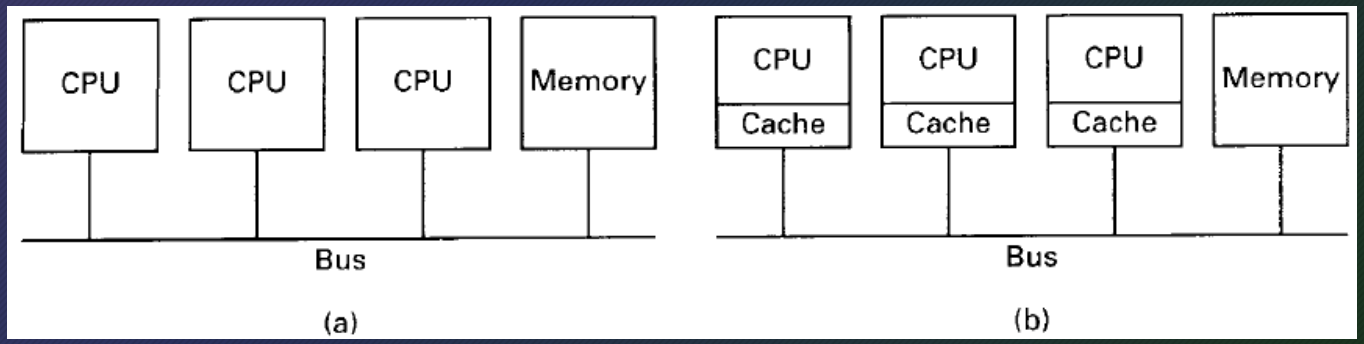
1. On-Chip Memory (contd…)



Types of DSM (contd…)

2. Bus-based Multiprocessors

- A set of parallel wires called a bus acts as a connection between CPU and memory
- Accessing of same memory simultaneously by multiple CPUs is prevented by using some algorithms
- Cache memory is used to reduce network traffic



(a) A multiprocessor (b) A multiprocessor with caching

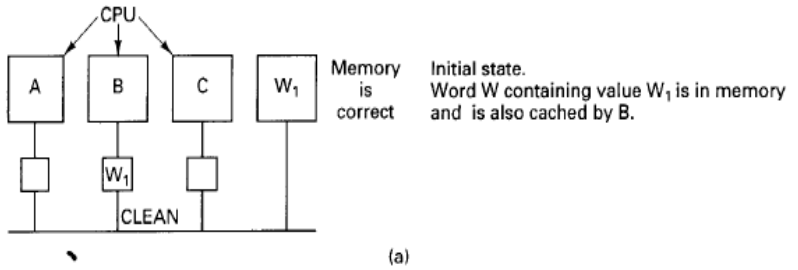
2. Bus-based Multiprocessors

Event	Action taken by a cache in response to its own CPU's operation	Action taken by a cache in response to a remote CPU's operation
Read miss	Fetch data from memory and store in cache	(No action)
Read hit	Fetch data from local cache	(No action)
Write miss	Update data in memory and store in cache	(No action)
Write hit	Update memory and cache	Invalidate cache entry

- Cache blocks can be in one of the following three states:

1. INVALID – This cache block does not contain valid data.
2. CLEAN – Memory is up-to-date; the block may be in other caches.
3. DIRTY – Memory is incorrect; no other cache holds the block.

2. Bus-based Multiprocessors (contd...)



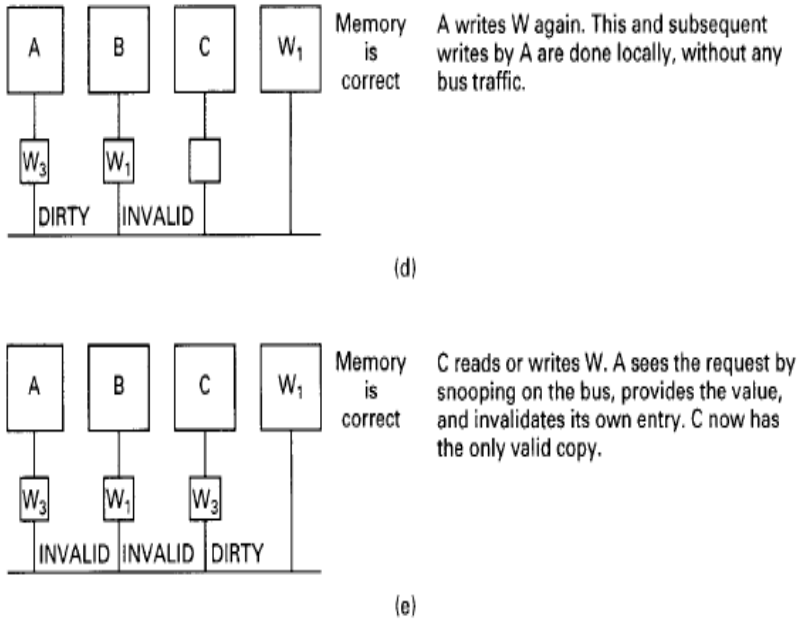


Fig. 6-4. An example of how a cache ownership protocol works.

Types of DSM (contd…)

3. Ring-based Multiprocessors
   - There is no global centralized memory present in Ring-based DSM
   - All nodes are connected via a token passing ring
   - In ring-based DSM a single address line is divided into the shared area

- Valid bit– whether the block is present in the cache and up to date.
- Exclusive bit– whether the local copy, if any, is the only one.
- Home bit-which is set only if this is the block's home machine.
- Interrupt bit- used for forcing interrupts.
- Location field– where the block is located in the cache if it is present and valid.

3. Ring-Based Multiprocessor

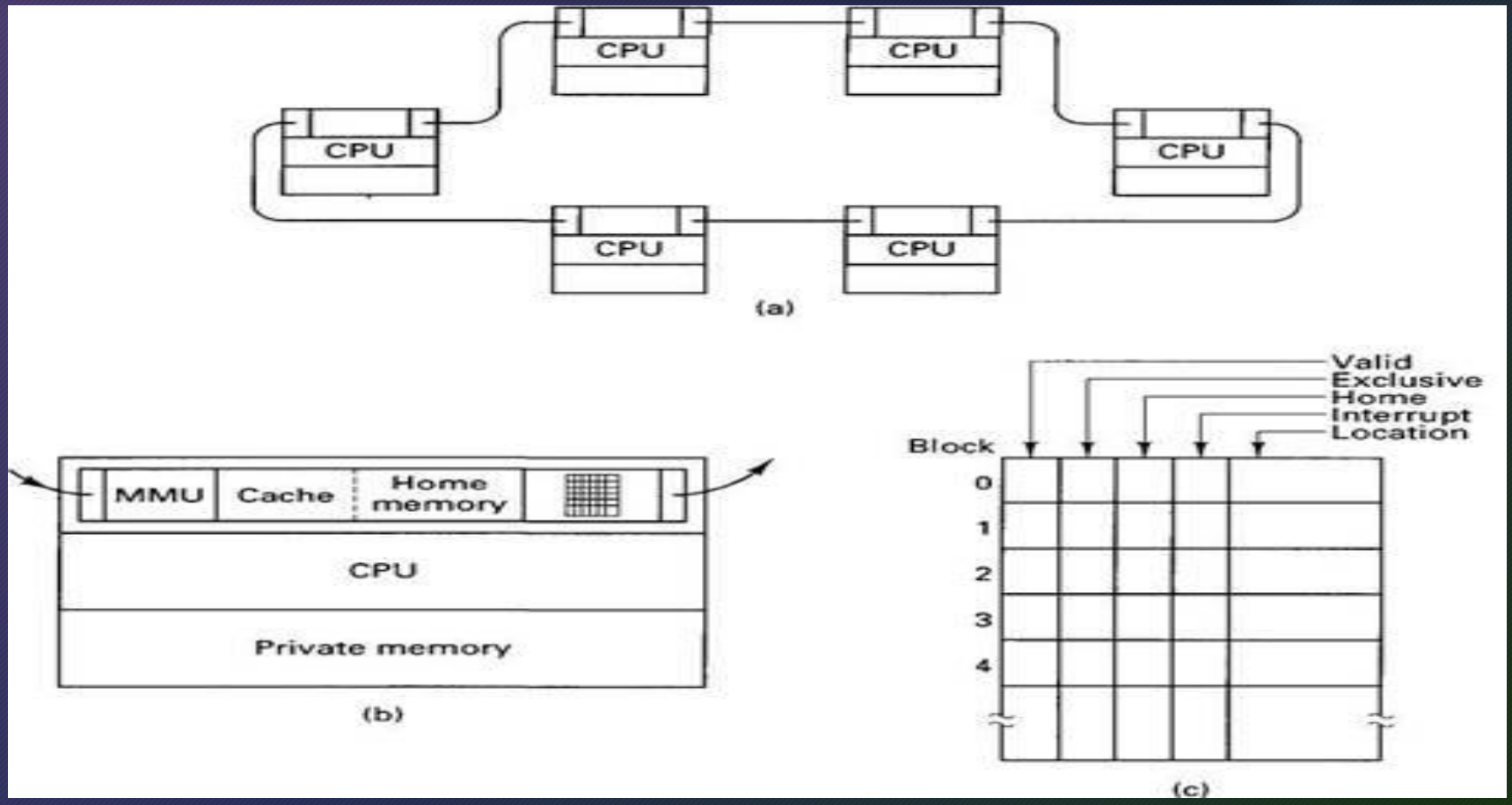


Fig (a) The memnet ring (b) A single machine (c) The block table

4. Switched Multiprocessor

Why it is needed ?

- Bus-based and Ring-based multiprocessors work fine for small system (up to 64 CPUs). So, they do not scale well for systems with hundreds or thousands of CPUs. As CPUs are added, bandwidth saturates at some point.

Possible Solution Approaches

1. **Reduce the amount of communication**: Already achieved this using caching. Additional work may include improving cache protocol, optimizing the cache block size, and reorganizing the algorithm to increase the locality of memory reference.

2. **Increase the communication capacity:** Discuss in the next slides onwards.

Increase Communication Capacity

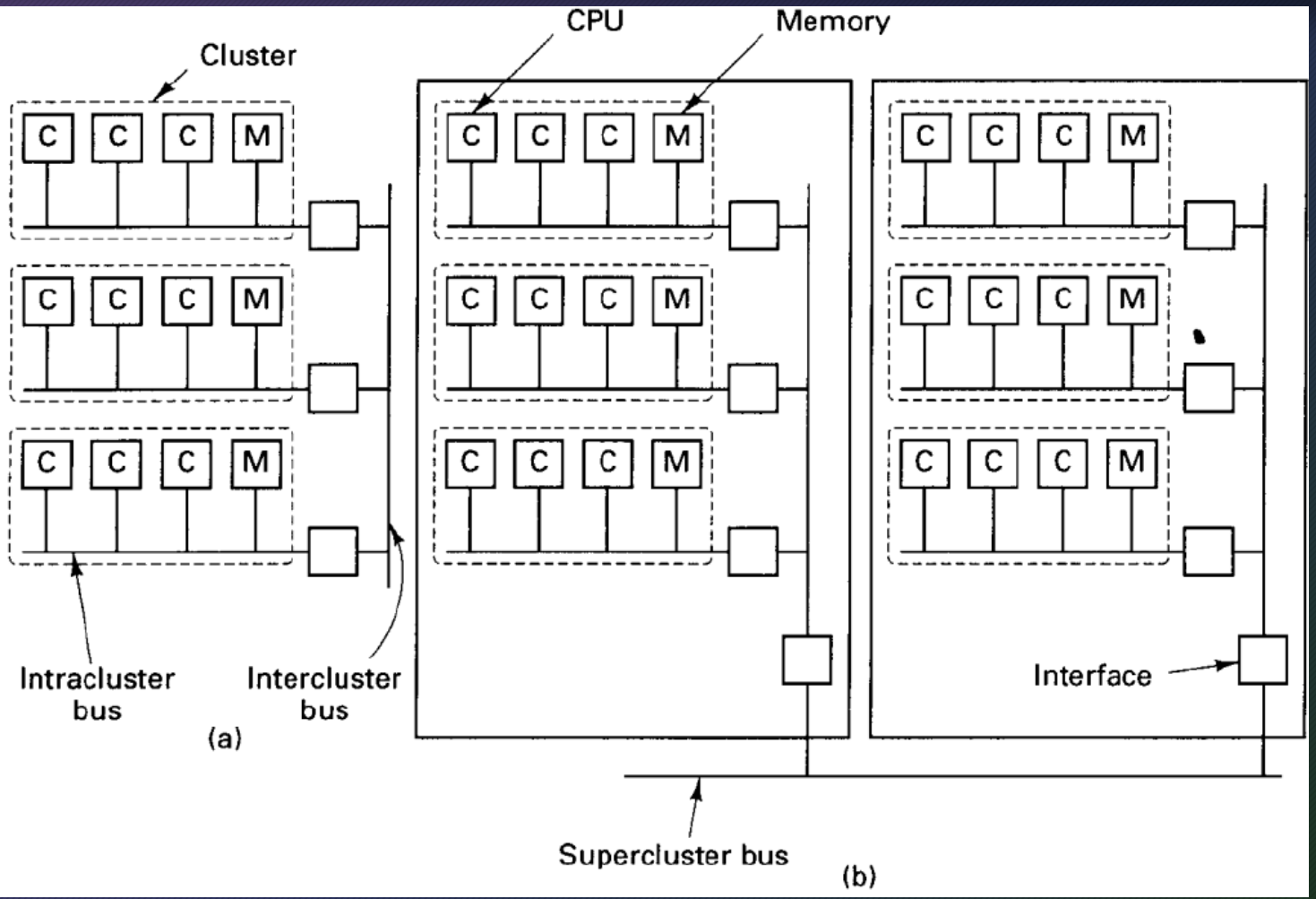
Approach 1: Change the topology

- Change topology from one bus to two buses or to a tree or grid. Changing the interconnection network's topology makes it possible to add additional capacity.

Approach 2: Build the system as a hierarchy

- Continue to put some number of CPUs on a single bus.
- Now consider this entire unit (CPUs + bus) as a cluster
- Build the system as multiple clusters and connect clusters using an inter-cluster bus (See Fig. (a) in the next slide).
- There will be relatively little traffic as long as most CPUs communicate primarily within their own cluster.
- If one inter-cluster bus proves to be inadequate, add a second inter-cluster bus or grid of clusters together into a supercluster and break the system into multiple supercluster.
- The supercluster can be connected by a bus, tree or a grid.(Fig. (b))

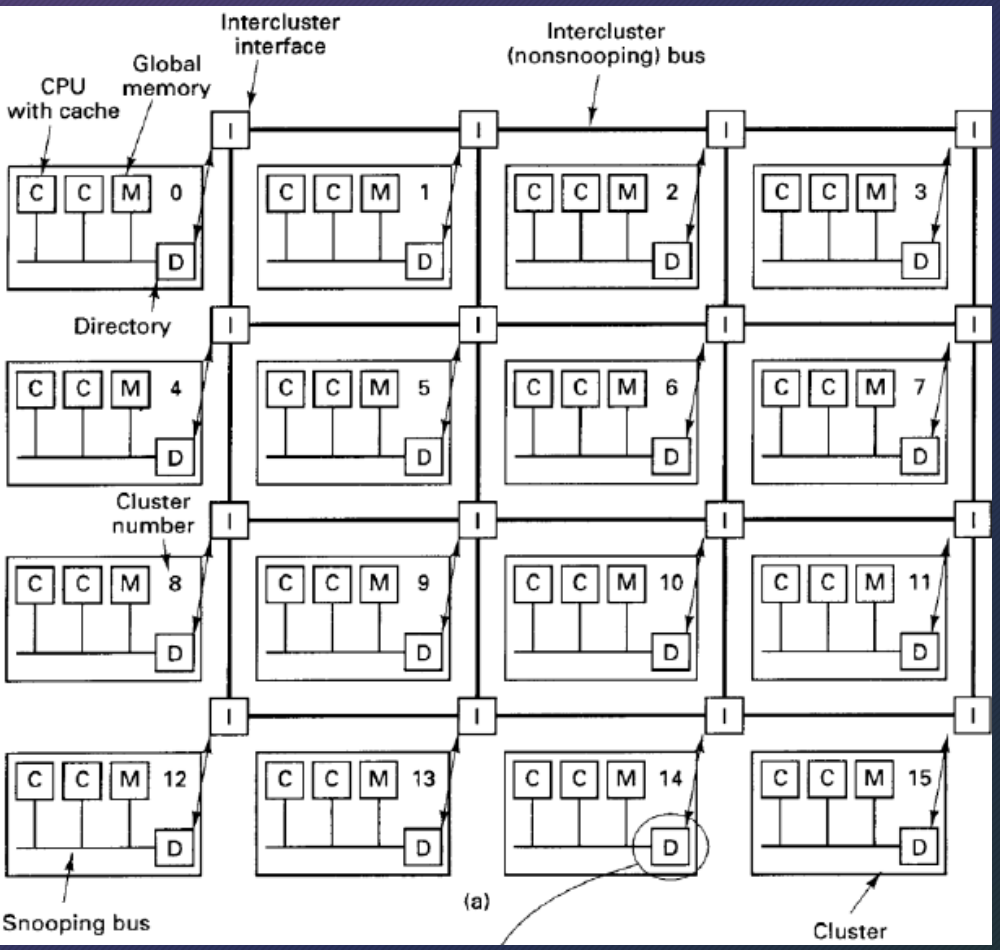
Design of Different Clusters



(a)Three clusters connected by an Inter-cluster bus from one supercluster

(b) Two superclusters connected by a supercluster bus

DASH: A Hierarchical Design based on a grid of Clusters



DASH: Directory Architecture for Shared Memory

It consists of 16 clusters, each cluster containing a bus, four CPUs, 16MB of the global memory and some I/O

(a) A simplified view of the DASH architecture. Each cluster actually has four CPUs, but only two are shown here

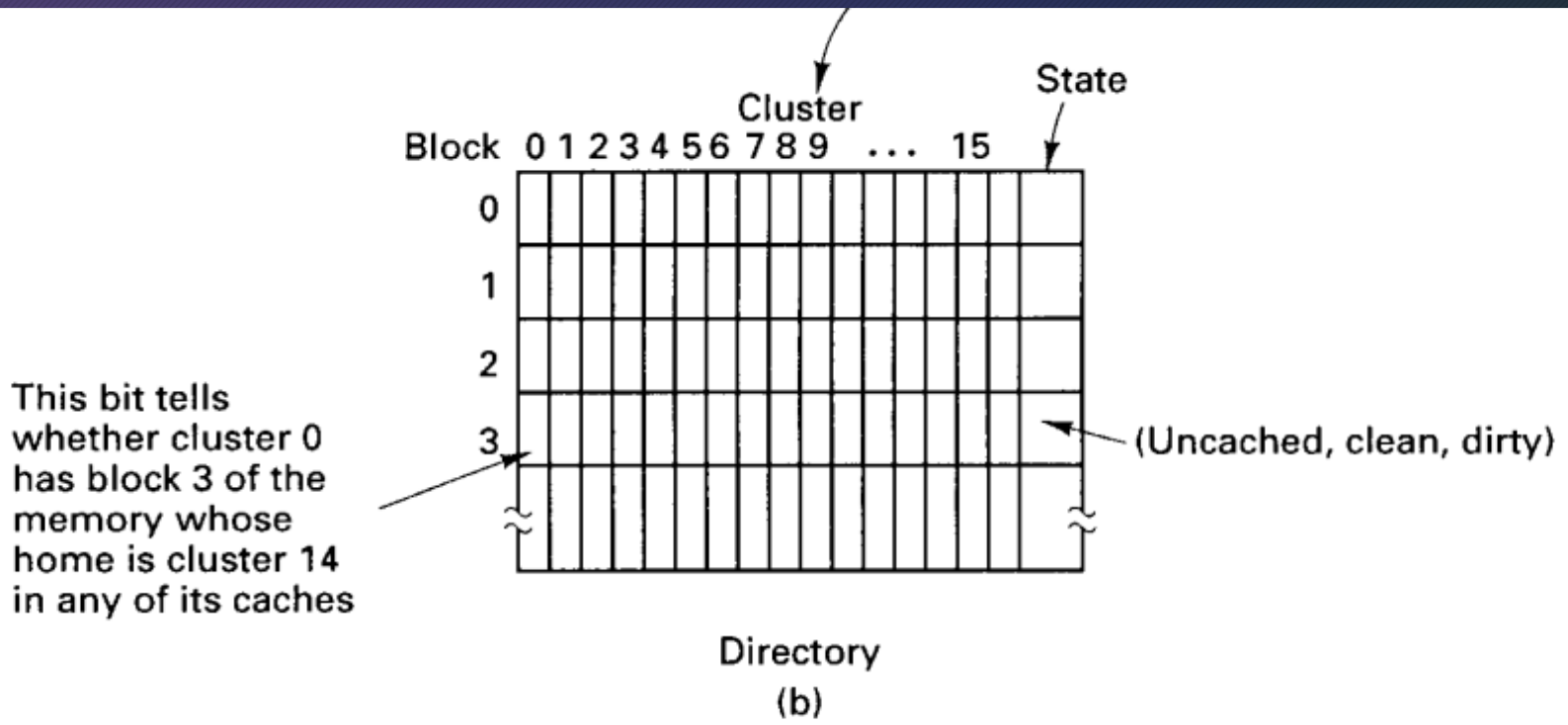
Directories

- Each cluster has a **directory** that keeps track of the clusters.
- 1M memory blocks in each cluster, it has 1M entries in the directory.
- Each entry holds a bit map with one bit per cluster tells whether or not the cluster has the block currently cached.
- The entry has also 2-bit field telling the state of the block.

Importance of Directory to the Operation of DASH

- Having 1M entries of 18 bits each means that the total size of each directory is over 2M bytes.
- With 16 clusters, the total directory memory about 14% of the 256M.
- If the number of CPUs per cluster is increased, the amount of directory memory is not changed.
- It allows the directory cost to be amortized over a large number of CPUs, reducing the cost per CPU.

Design of DASH Directories



Caching

- It is done in two levels; a first-level cache and a larger-level second cache.
- First-level cache is a subset of larger-level cache.
- Each cache block can be in any of the following states:
  1. UNCACHED: The only copy of the blocks in the memory
  2. CLEAN: Memory is up-to-date; the block may be in several cache.
  3. DIRTY: The memory is incorrect; only one cache holds the block.
- The state of each cache block is stored in the Store field of its directory entry (shown in previous slide).

Protocols

Protocols

DASH Protocol: Based on ownership and Invalidation

- At every instance, each cache block has a unique owner.
- For UNCACHED and CLEAN, the block's home cluster is owner.
- For DIRTY blocks, the cluster holding the one and only copy is owner.
- Writing on a CLEAN block requires first finding and invalidating all the existing copies. This is where the directory comes in.

How CPU reads a memory word ?

- It first checks its own caches.
- If neither cache has the word, a request is issued on the local cluster bus to see if another CPU in the cluster has the block containing it.
- If one does, a cache-to-cache transfer of the block is executed to place the block in the requesting CPU
- If the block is CLEAN, a copy is made; If it is DIRTY, the home directory is informed that the block is now CLEAN and shared.

Protocols (contd…)

Process of Cache Read

- If the block is not present in any of the cluster's cache, a request packet is sent to the block's home cluster.

How CPU writes a memory word ?

- Before a write can be done, CPU must ensure that it is the owner of the only copy of the cache block in the system.
- If it already has the blocks in its on-board cache and the block is DIRTY, the write can proceed immediately.
- If it has the block but it is CLEAN, all other copies must be invalidated by the home cluster.

Directory Architecture for Shared memory (DASH)

- Distributed shared memory paradigm enables shared memory view of a loosely coupled distributed memory system. The DASH prototype developed at Stanford University is example of such systems.

- The DASH prototype belongs to the non uniform memory access (NUMA) class and makes use of the directory-based protocol for maintaining cache coherence.

- The DASH is a high-performance machine with single address space and coherent caches.

- The DASH architecture consists of a 2-D mesh network of clusters; each cluster consists of a set of processor elements (PEs) sharing a common communication channel. The cache coherence protocolis based on a four-level memory hierarchy protocol.

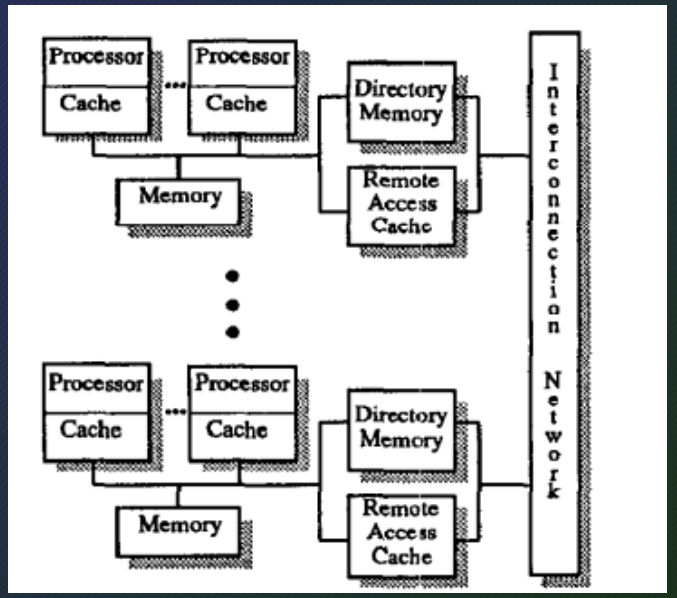


Figure 1: General architecture of DASH.

DASH Cache Coherence Protocol

- The DASH coherence protocol is an invalidation-based ownership protocol.
- A memory block can be in one of three states as indicated by the associated directory entry:
  (i) uncached-remote, that is not cached by any remote cluster
  (ii) shared-remote, that is cached in an unmodified state by one or more remote clusters or
  (iii) dirty-remote, that is cached in a modified state by a single remote cluster
- There are three primitive operations supported by the base DASH coherence protocol i.e., read, read-exclusive and write-back.

  Read Requests: Memory read requests are initiated by processor load instructions. If the location is present in the processor's first-level cache, the cache simply supplies the data. If not present, then a cache fill operation must bring the required block into the first-level cache.

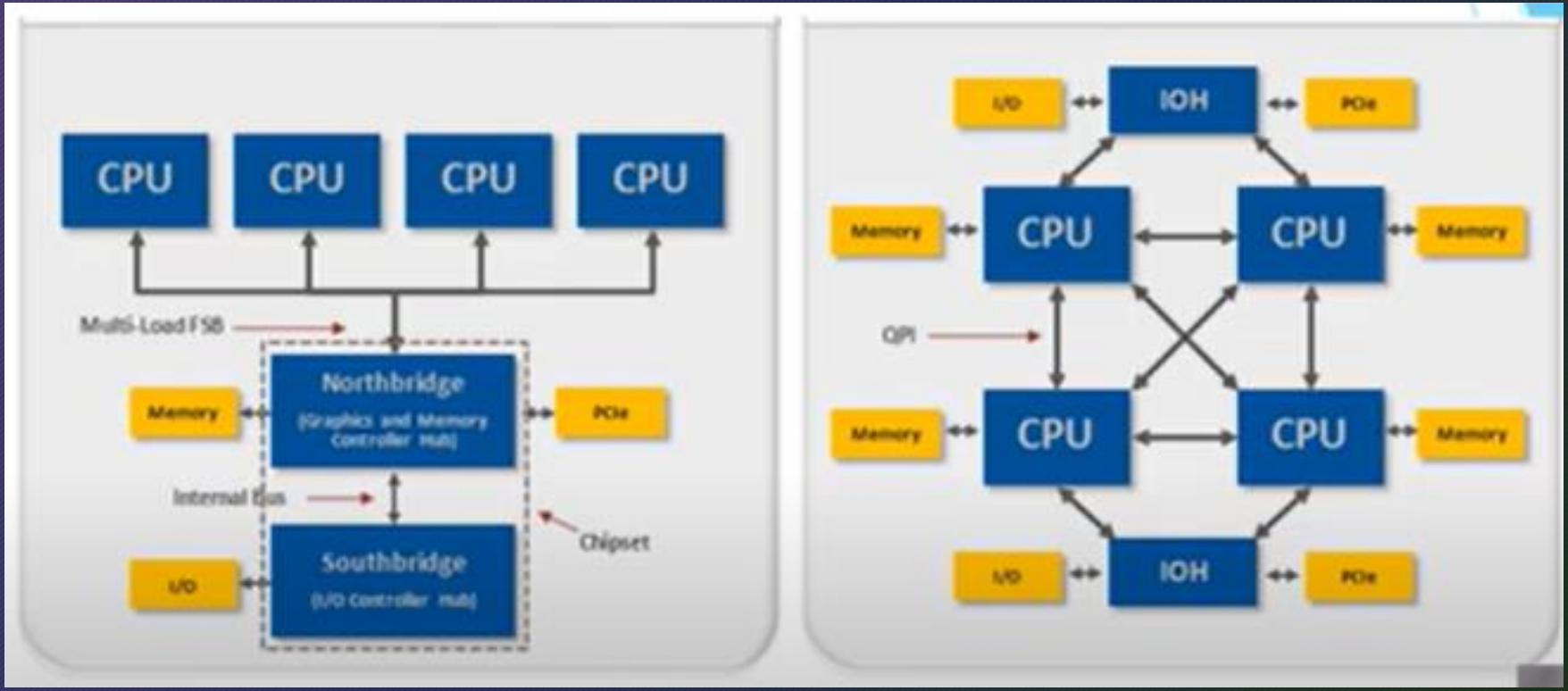
  Read-Exclusive Requests: Write operations are initiated by processor store instructions. Data is written through the first-level cache and is buffered in a four word deep write-buffer. The second-level cache can retire the write if it has ownership of the line. Otherwise, a read exclusive request is issued to the bus to acquire sole ownership of the line and retrieve the other words in the cache block.

  Writeback Requests: A dirty cache line that is replaced must be written back to memory. If the home of the memory block is the local cluster, then the data is simply written back to main memory. If the home cluster is remote, then a message is sent to the remote home which updates the main memory and marks the block uncached-remote.

Non-uniform memory access (NUMA)

- Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing systems where memory access time depends on the memory location relative to the processor accessing it.
- In a NUMA architecture, multiple processors (or nodes) are connected to a shared memory pool.
- Each processor has its own local memory and can also access memory from other processors over a high-speed interconnect.
- However, accessing local memory is faster than accessing remote memory due to differences in latency.
- The primary motivation behind NUMA is to alleviate memory access bottlenecks in large-scale multiprocessing systems. As the number of processors increases in a system, the contention for accessing a shared memory pool also increases, leading to performance degradation.
- By partitioning memory into local and remote segments based on the proximity to processors, NUMA aims to reduce contention and improve overall system performance.
- NUMA architectures typically employ hardware or software mechanisms to manage memory access efficiently.

UMA vs NUMA



UMA: Uniformed Memory Access

NUMA: Non-Uniformed Memory Access

NUMA Algorithms

- **Linux NUMA Balancing:** In modern Linux systems, the kernel tracks memory access patterns and automatically moves memory pages closer to the processors accessing them, which is referred to as automatic NUMA balancing. It employs algorithms to determine whether migrating memory pages would improve performance and make memory allocation decisions accordingly.

- **Windows NUMA Scheduling and Allocation:** In Windows, NUMA-aware scheduling and memory management algorithms allow applications and the OS to optimize memory access patterns based on NUMA node locality, improving the performance of NUMA-enabled systems.

- **Java Virtual Machine (JVM) NUMA Awareness:** In memory-managed environments like the JVM, garbage collection algorithms can be NUMA-aware, ensuring that memory allocations and deallocations respect memory locality. This helps in reducing remote memory accesses and optimizing performance for memory-intensive applications.

Importance of NUMA

NUMA is essential to modern servers and mainframe environments for several reasons. This is because it paves the way for:

- Reduced memory access latency
- Improved performance
- Greater flexibility
- Enhanced parallelism
- Optimized cache utilization
- Adaptive resource management
- Fault tolerance
- High availability
- Compatibility with heterogeneous workloads
- Energy efficiency

Applications of NUMA

NUMA architecture finds applications across various industries requiring scalable multiprocessing systems to handle complex tasks efficiently. Here are some use cases where NUMA-based solutions can be presented.

- **Enterprise computing:** In enterprise computing environments, NUMA architectures are widely used in servers and data centers to support mission-critical applications such as database management systems (DBMS), enterprise resource planning (ERP) software, and virtualization platforms.
- **High-performance computing (HPC):** In scientific research, engineering simulations, and computational modeling, NUMA models play a crucial role in high-performance computing (HPC) clusters and supercomputers.
- **Financial services:** NUMA frameworks are used in trading platforms, risk analysis systems, and algorithmic trading algorithms in the financial services industry.
- **Telecommunications:** NUMA systems are employed in network appliances, packet processing systems, and software-defined networking (SDN) controllers in telecommunications infrastructure. Telcos require fast and efficient data processing to handle network traffic routing, quality of service (QoS) management, and network function virtualization (NFV).

Applications of NUMA (contd...)

- **Healthcare and life sciences:** In healthcare and life sciences research, NUMA architectures support applications such as medical imaging analysis, genomic sequencing, and drug discovery.
- **Aerospace and defense:** NUMA is necessary for simulation and modeling systems for aircraft design, missile guidance systems, and radar signal processing. This makes it possible to simulate complex aerodynamic phenomena, analyze sensor data in real time, and optimize defense systems' performance.
- **Automotive:** NUMA architectures are utilized in vehicle design and testing, autonomous driving systems, and manufacturing process optimization. Automotive manufacturers rely on computational simulations to design and validate vehicle components, analyze crash scenarios, and optimize fuel efficiency.
- **Ecommerce and retail:** In the e-commerce and retail industry, NUMA mechanisms power recommendation engines, inventory management systems, and supply chain optimization algorithms.

Page-based Distributed Shared Memory

Page-based Distributed Shared Memory

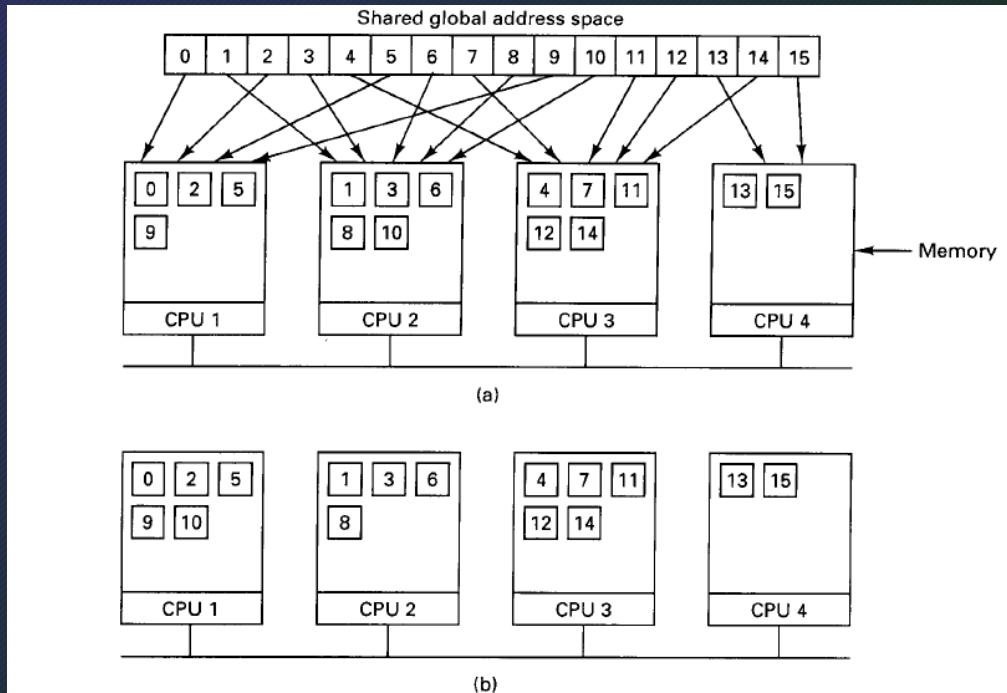
- IVY-System: These systems are built on top of multicomputers that is, processors connected by a specialized message-passing network, workstations on a LAN, or similar designs. The essential element here is that no processor can directly access any other processor's memory. Such systems are sometimes called NORMA (NO Remote Memory Access) systems to contrast them with NUMA systems.
- At hardware level memory access of other CPU is not possible in NORM.
- Chip design is different.
- In NUMA, at hardware level CPU and memory are integrated (bundled). The memory access of CPU for local memory is faster compared to the memory access of other CPU, therefore NUMA is called non-uniform memory accessed.
- NUMA
  - Access remotely – no copying page into local memory, it is needed for temporary usage.
  - Fetch – copying actual page into local memory.
- NORMA
  - No remote access.
  - Fetch is mandatory.

Basic Design

In a DSM system, the address space is divided up into chunks, with the chunks being spread over all the processors in the system. When a processor references an address that is not local, a trap occurs, and the DSM software fetches the chunk containing the address and restarts the faulting instruction, which now completes successfully.

This concept is illustrated in Fig. 6-25(a) for an address space with 16 chunks and four processors, each capable of holding four chunks.

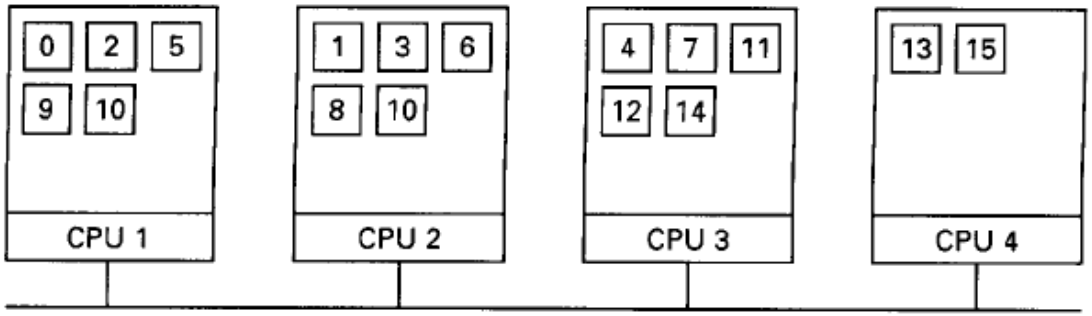
In this example, if processor 1 references instructions or data in chunks 0, 2,5, or 9, the references are done locally. References to other chunks cause traps. For example, a reference to an address in chunk 10 will cause a trap to the DSM software, which then moves chunk 10 from machine 2 to machine 1, as shown in Fig. 6-25(b).



Replication

One improvement to the basic system that can improve performance considerably is to replicate chunks that are read only, for example, program text, read-only constants, or other read-only data structures. For example, if chunk 10 in Fig. 6-25 is a section of program text, its use by processor 1 can result in a copy being sent to processor 1, without the original in processor 2's memory being disturbed, as shown in Fig. 6-25(c). In this way, processors 1 and 2 can both reference chunk 10 as often as needed without causing traps to fetch missing memory.

Another possibility is to replicate not only read-only chunks, but all chunks. As long as reads are being done, there is effectively no difference between replicating a read-only chunk and replicating a read-write chunk. However, if a replicated chunk is suddenly modified, special action has to be taken to prevent having multiple, inconsistent copies in existence.



(c)

Fig. 6-25. (a) Chunks of address space distributed among four machines. (b) Situation after CPU 1 references chunk 10. (c) Situation if chunk 10 is read only and replication is used.

Granularity

At what level granularity, user wants to access the data.

- Accessing a word.
- Accessing a page.
- Accessing a block.

There is an issue with granularity is **FALSE SHARING**.

- A page has two unrelated variables; both variables are needed by more than one CPU; then false sharing happens, performance will be an issue in such situation.
- Clever compilers keep same page in more than one memory location to avoid false sharing and improve performance.
- Clever compilers cannot solve false sharing problem for arrays, structures, unions, data structures and classes, since collection of data in similar or dissimilar format in multiple contiguous or non-contiguous locations.

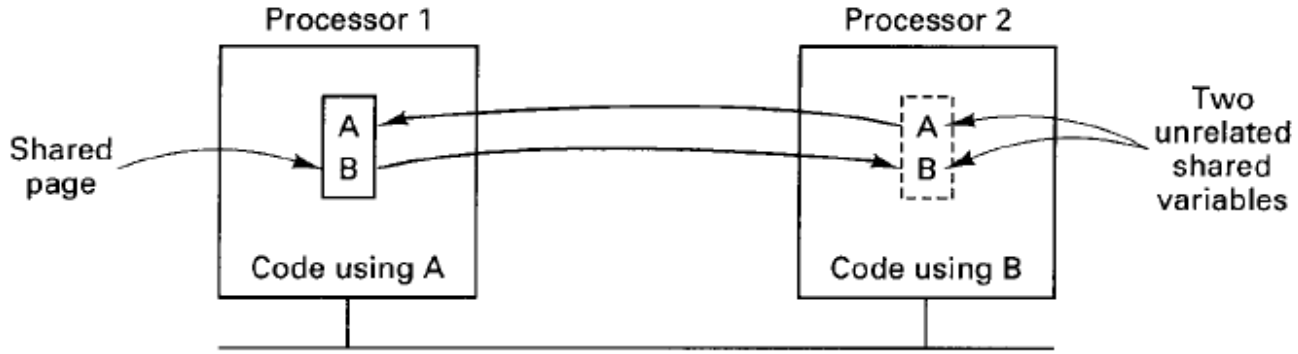
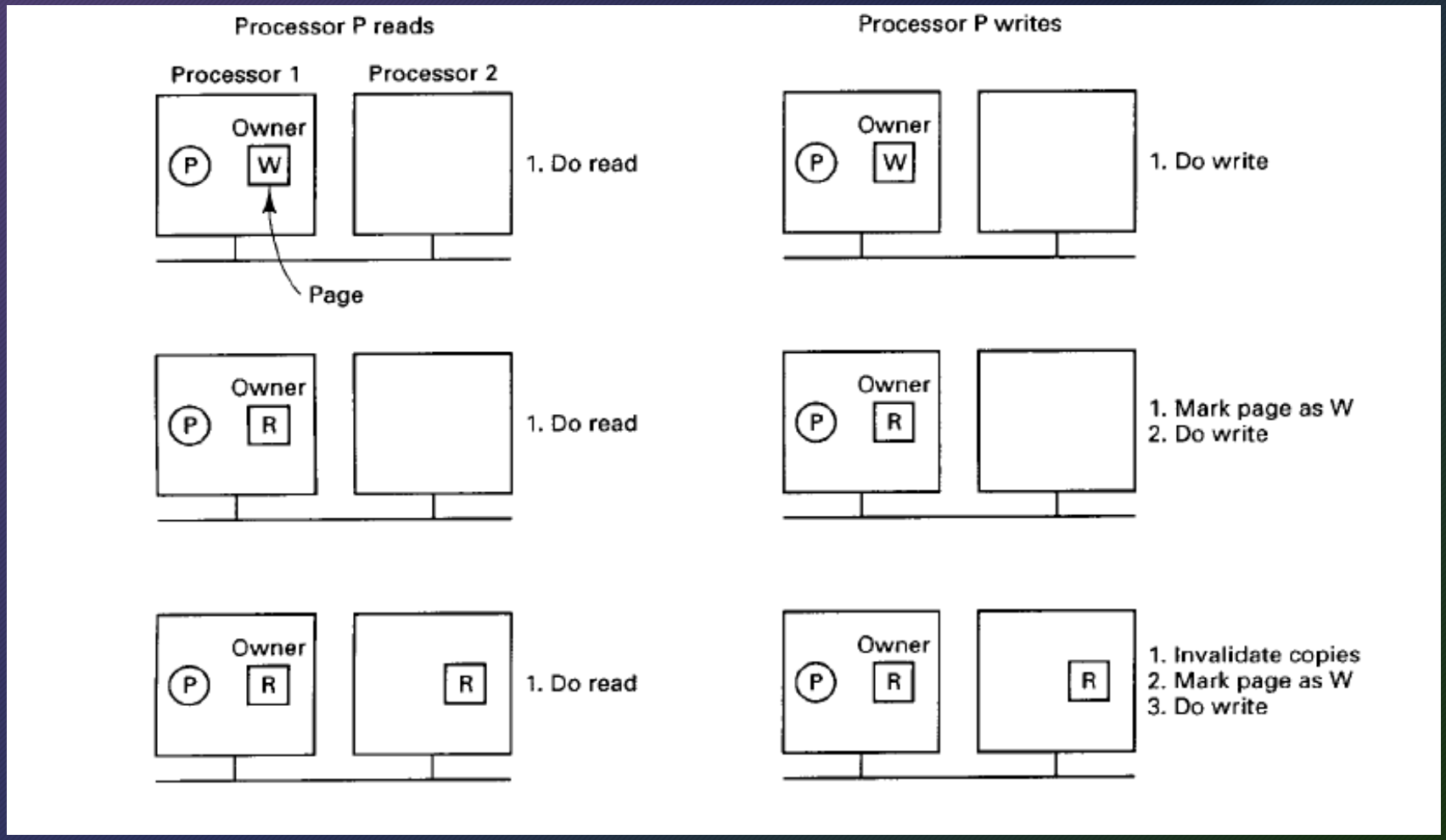


Fig. 6-26. False sharing of a page containing two unrelated variables.

Achieving Sequential Consistency



Achieving Sequential Consistency (contd…)

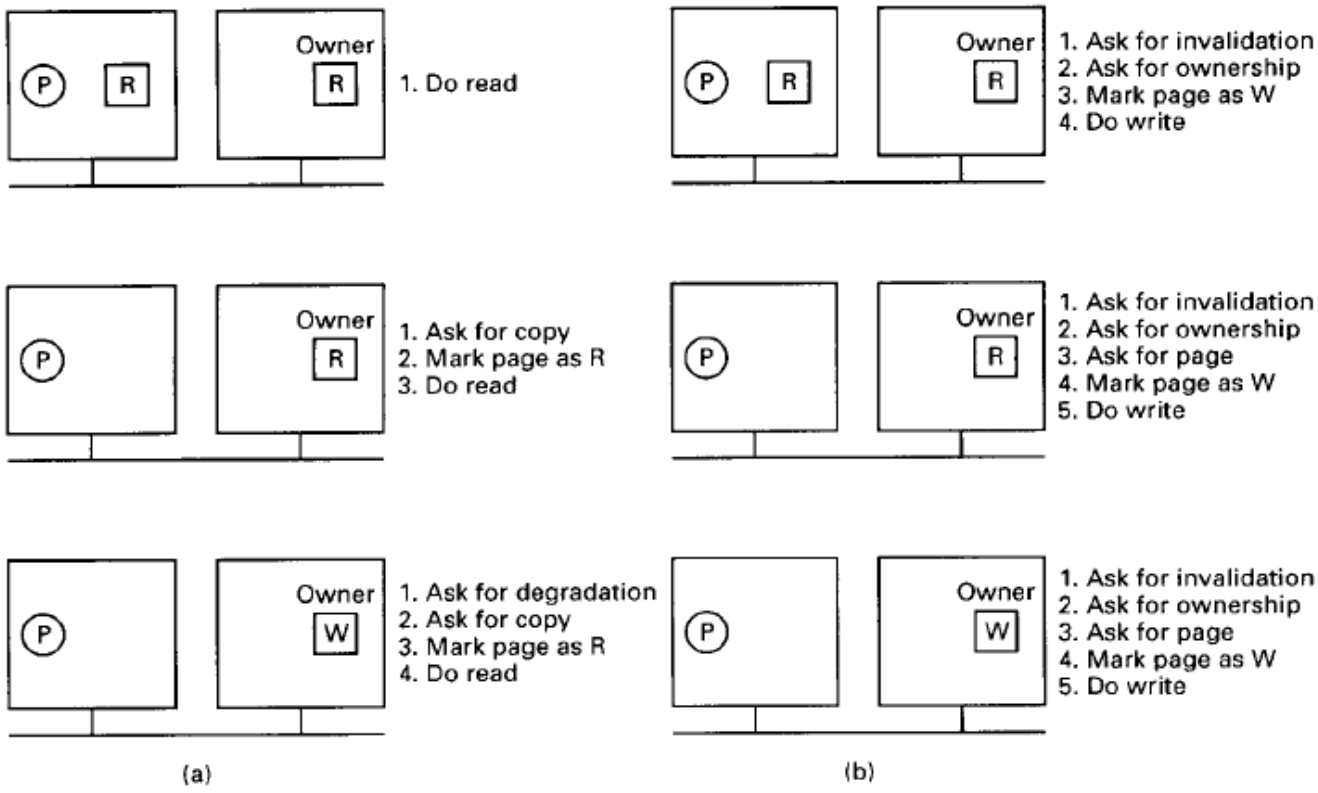


Fig. 6-27. (a) Process *P* wants to read a page. (b) Process *P* wants to write a page.

Finding the Owner

- One of them is how to find the owner of the page. The simplest solution is by doing a broadcast, asking for the owner of the specified page to respond. Once the owner has been located this way, the protocol can proceed as above.
- An obvious optimization is not just to ask who the owner is, but also to tell whether the sender wants to read or write and say whether it needs a copy of the page. The owner can then send a single message transferring ownership and the page as well, if needed.
- Broadcasting has the disadvantage of interrupting each processor, forcing it to inspect the request packet. For all the processors except the owner's, handling the interrupt is essentially wasted time. Broadcasting may use up considerable network bandwidth, depending on the hardware.

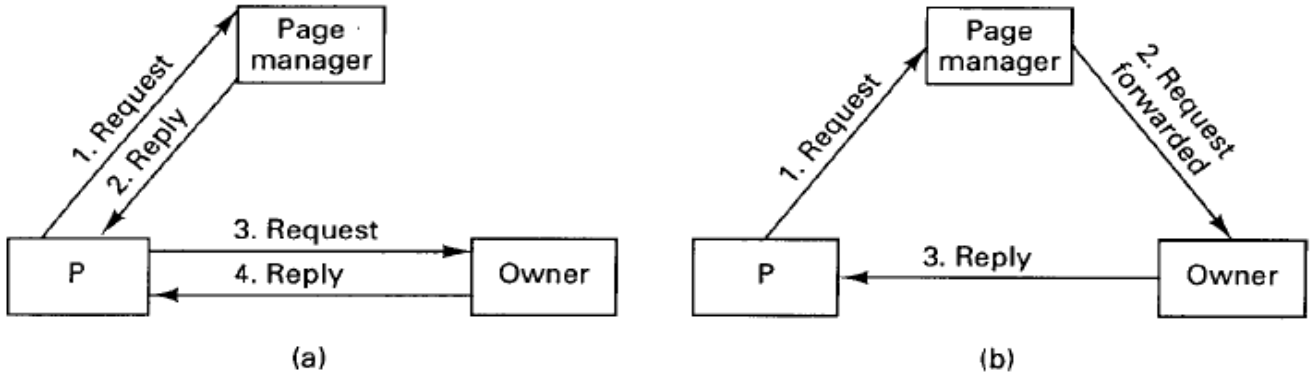


Fig. 6-28. Ownership location using a central manager. (a) Four-message protocol. (b) Three-message protocol.

Finding the Owner (contd…)

- Li and Hudak (1989) describe several other possibilities as well. In the first of these, one process is designated as the page manager. It is the job of the manager to keep track of who owns each page. When a process, P, wants to read a page it does not have or wants to write a page it does not own, it sends a message to the page manager telling which operation it wants to perform and on which page. The manager then sends back a message telling who the owner is.
- P now contacts the owner to get the page and/or the ownership, as required. Four messages are needed for this protocol, as illustrated in Fig. 6-28(a).

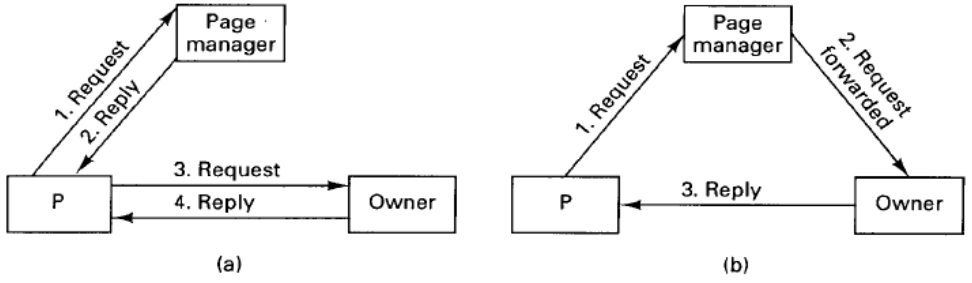


Fig. 6-28. Ownership location using a central manager. (a) Four-message protocol. (b) Three-message protocol.

Finding the Owner (contd…)

- An optimization of this ownership location protocol is shown in Fig. 6-28(b). Here the page manager forwards the request directly to the owner, which then replies directly back to *P*, saving one message.
-
- A problem with this protocol is the potentially heavy load on the page manager, handling all the incoming requests. This problem can be reduced by having multiple page managers instead of just one. Splitting the work over multiple managers introduces a new problem, however—finding the right manager.

Finding the Owner (contd…)

- **A simple solution** is to use the low-order bits of the page number as an index into a table of managers. Thus with eight page managers, all pages that end with 000 are handled by manager 0, all pages that end with 001 are handled by manager 1, and so on. A different mapping, for example by using a hash function, is also possible. The page manager uses the incoming requests not only to provide replies but also to keep track of changes in ownership. When a process says that it wants to write on a page, the manager records that process as the new owner.

- **Still another possible algorithm** is having each process (or more likely, each processor) keep track of the probable owner of each page. Requests for ownership are sent to the probable owner, which forwards them if ownership has changed. If ownership has changed several times, the request message will also have to be forwarded several times. At the start of execution and every n times ownership changes, the location of the new owner should be broadcast, to allow all processors to update their tables of probable owners.

- **The problem of locating the manager** also is present in multiprocessors, such as Dash, and also in Memnet. In both of these systems it is solved by dividing the address space into regions and assigning each region to a fixed manager, essentially the same technique as the multiple-manager solution discussed above, but using the high-order bits of the page number as the manager number.

Finding the Copies

- Another important detail is how all the copies are found when they must be invalidated. Again, two possibilities present themselves. The first is to broadcast a message giving the page number and ask all processors holding the page to invalidate it. This approach works only if broadcast messages are totally reliable and can never be lost.
- The second possibility is to have the owner or page manager maintain a list or copyset telling which processors hold which pages, as depicted in Fig. 6-29. Here page 4, for example, is owned by a process on CPU 1, as indicated by the double box around the 4. The copyset consists of 2 and 4, because copies of page 4 can be found on those machines.
- When a page must be invalidated, the old owner, new owner, or page manager sends a message to each processor holding the page and waits for an acknowledgement. When each message has been acknowledged, the invalidation is complete.

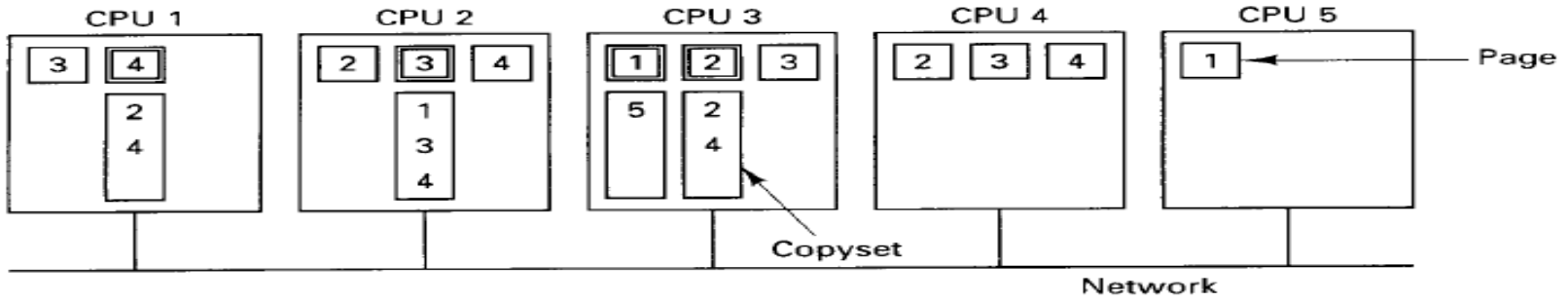


Fig. 6-29. The owner of each page maintains a copyset telling which other CPUs are sharing that page. Page ownership is indicated by the double boxes.

Page Replacement

- When memory is less than number of pages available; then page replacement is required.
- Three different cases to identify which page to be replaced :
  1. You are the owner, but no body holds any copy of the same page.
  2. You are the owner, copies exist with other CPU's.
  3. You are not a owner, but holding the copy of the page; somebody else is owner.
- Priority:
  - First priority is case (3); i.e., drop the pages for whom you are not owner.
  - Second priority is case (2); simply change ownership and notify coordinator(if it is exists); otherwise broadcast to all that you are changing the ownership of the page (in case of no coordinator exist), then drop the page for replacing other page(s).
  - Third priority is case (1); send this page to some other CPU's or store it in hard disk and mark it is passive, so that space is created in the main memory to replace other important pages.

- Synchronization
- Mutual exclusion : TEST-AND-SET-LOCK (TSL) instruction is often used to implement mutual Exclusion. In normal use, a variable is set to 0 when no process is in the critical section and to 1 when one process is. The TSL instruction reads out the variable and sets it to 1 in a single, atomic operation. If the value read is 1, the process just keeps repeating the TSL instruction until the process in the critical region has exited and set the variable to 0.

Shared-Variable Distributed Shared Memory

Shared Variable Distributed Shared Memory

- In this scheme, we are not sharing entire page, instead we are sharing primitives, data structures or variables. Transfer level is reduced.
- **MUNIN DSM**
- Granularity can be solved using clever compiler, which is residing with MUNIN.
- MUNIN can place each object on a separate page, so that hardware MMU can be used for detecting accesses to shared objects. The basic model used by MUNIN is that of multiple processors, each with a paged linear address space in which one or more threads are running a slightly modified multiprocessor program (Quad Core).
- It is like dusty deck problem(Dusty deck problem – old technique or methodologies are reused; instead of writing new code.), i.e., multiprocessor program is existing is reused and slight modifications are done. The slight modifications are done using ANNOTATIONS.
- E.g., In JAVA@OVERRIDE i.e., method overriding , i.e., pass instructions to compiler using @ operator and add additional program to embed in multiprocessor program. This is done using keyword SHARED. In other languages it may be @ or # used for reusing existing code.

- **Release Consistency**
- Three kinds of variables are used in MUNIN

1. Ordinary Variable – local to process i.e., auto in C language. It is not shareable, not visible to other processes.
2. Shared data Variable – by lock and unlock critical section only one process can use and update.
3. Synchronized Variable – Lock(L) unlock(L) – L is synchronized variable, it is not data.

Multiple Protocols

1. Read only – No write operations; multiple copies exist.
2. Migratory – shared variables are migratory Fig 6-30.
3. Write-shared – Railway reservation; where many counters can write into a database, only one copy- Fig 6-31. We create backup like Fig 6-31 – we create TWIN and make RW, in RW we are updating the new value by issuing write trap; then release after updating and inform all other CPU that this modifications are done; once they confirm updation then we complete the task, otherwise RUNTIME error is generated; since all CPU's have not updated new value(i.e., 8) and rollback to original value(i.e., 6).
4. Conventional – opposite to read only – writeable; multiple copies cannot exists; it can be made available in multiple based on demand after closing file.

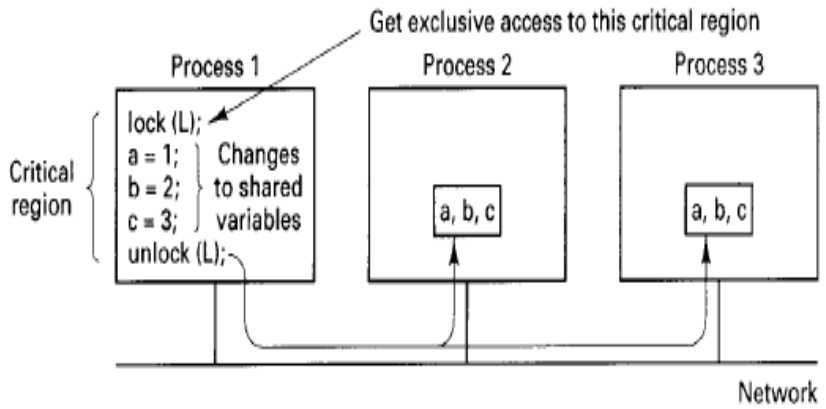


Fig. 6-30. Release consistency in Munin.

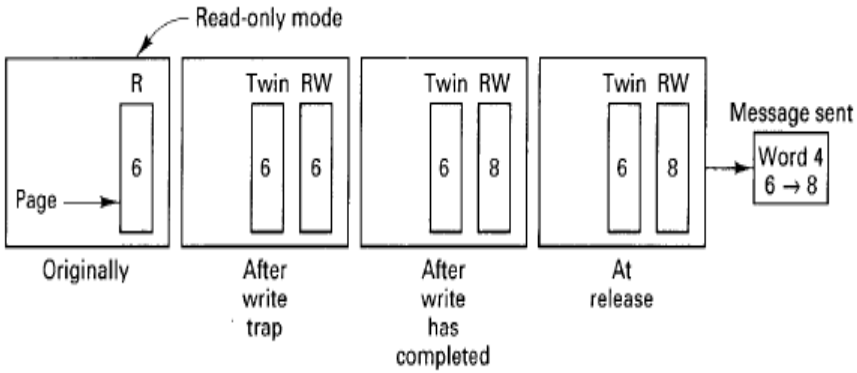


Fig. 6-31. Use of twin pages in Munin.

Multi-writer Protocols

- Fig 6-32(c) and (d) process p1 and p2 are run on 2 different CPU's even series and odd series updation are done by 2 CPU's. In Fig (c) if a[0] has done updation then control goes to CPU (2) to update a[1]; so the message passing time is more and wastage of time. It is worse than uniprocessor.

- Whereas in Fig (d) 2 CPU's do computations parallely and inform and exchange messages at the end, speed is increased by 50%. If 2 CPU's are used. If 3 CPU's are used then each CPU may take 33%, so performance gain (speed) is 66%.

```
Process 1                          Process 2

/* Wait for process 2 */           /* Wait for process 1 */
wait_at_barrier(b);                wait_at_barrier(b);

for (i = 0; i < n; i +=2)          for (i = 1; i < n; i +=2)
    a[i] = a[i] + f(i);                a[i] = a[i] + g(i);

/* Wait until proc 2 is done */   /* Wait until proc 1 is done */
wait_at_barrier(b);                wait_at_barrier(b);
```

(a) (b)

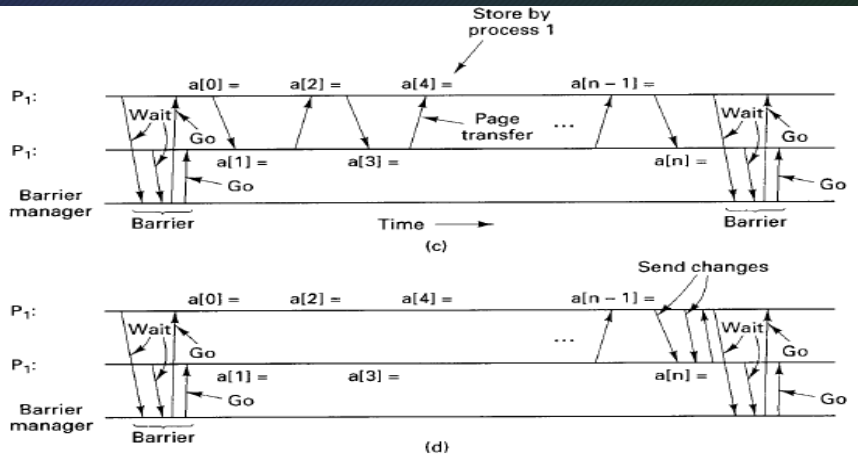


Fig. 6-32. (a) A program using *a*. (b) Another program using *a*. (c) Messages sent for sequentially consistent memory. (d) Messages sent for release consistent memory.

Object-Based Distributed Shared Memory

Object-Based DSM

- **Object-based:** it comprises of objects, classes, encapsulation, abstraction and polymorphism, but it will not have inheritance, message passing and dynamic binding. If it contains all these 3 then it will become object-oriented. We can enforce business rules using encapsulation.
- LINDA – it is a system; which will add additional small set of primitive operations to existing language such as C or Fortran to form parallel language C_LINDA or FORTRAN-LINDA.
- Tuple-Space – global space; multiple machines can insert / delete tuples.
- **Operations of Tuple:**
- Checkout- from our repository the data is written in global space database then it is checkout. It is done by **out function(parameters), e.g., out(parameters)** – to place new tuple in tuple space.
- Checkin – getting from global database to our repository. It is done by **in function(parameters), e.g., in(parameters)** – to get/retrieve tuple from tuple space.
- in("task.bag",?job) - ? is like reference to variable.
- LINDA is built over prolog. (uniprocessor). in ->pop out->push peep->read (only read from stack)
- eval -> evaluate (some calculation)

Implementation of Linda

An efficient Linda implementation has to solve two problems:

1. How to simulate associative addressing without massive searching.
2. How to distribute tuples among machines and locate them later.

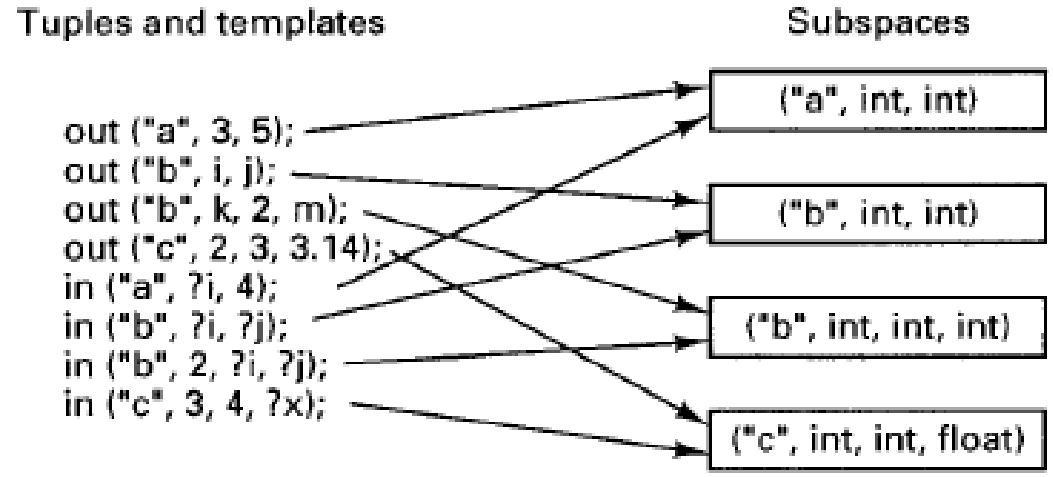


Fig. 6-37. Tuples and templates can be associated with subspaces.

Linda

1. Complete replication on each and every machine.

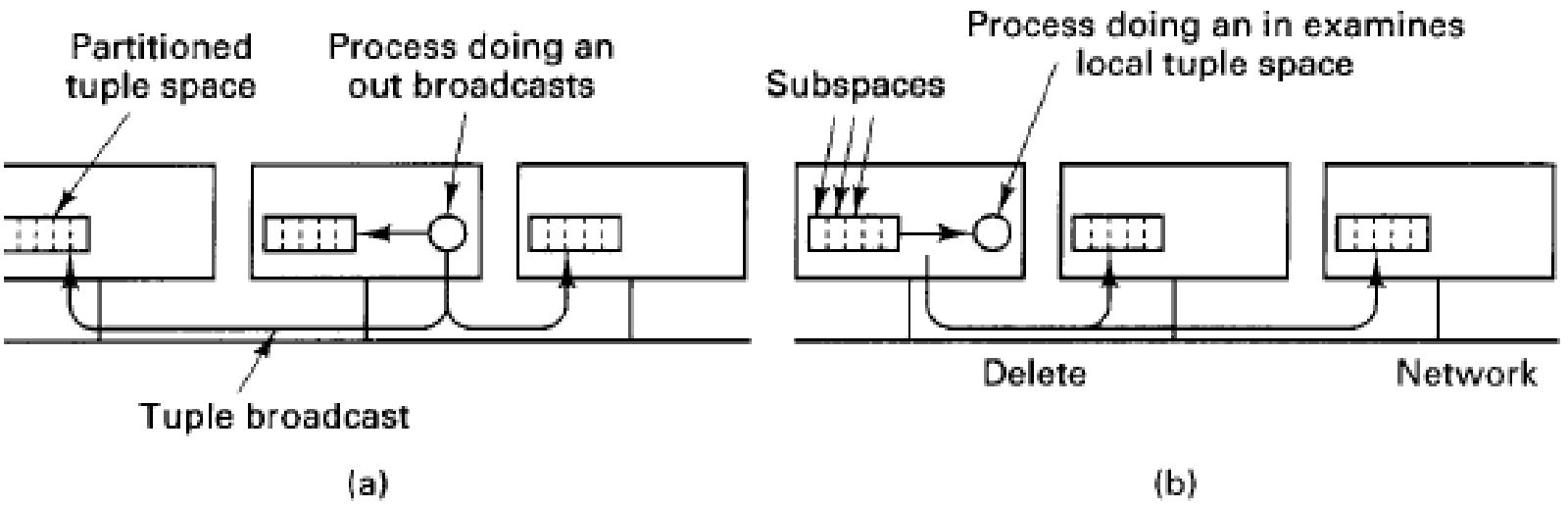


Fig. 6-38. Tuple space can be replicated on all machines. The dotted lines show the partitioning of the tuple space into subspaces. (a) Tuples are broadcast on *out*. (b). *In*s are local, but the deletes must be broadcast.

Linda (contd…)

2. No replication – independent replication of tuple space – deleting from one machine and replicating in another machine.

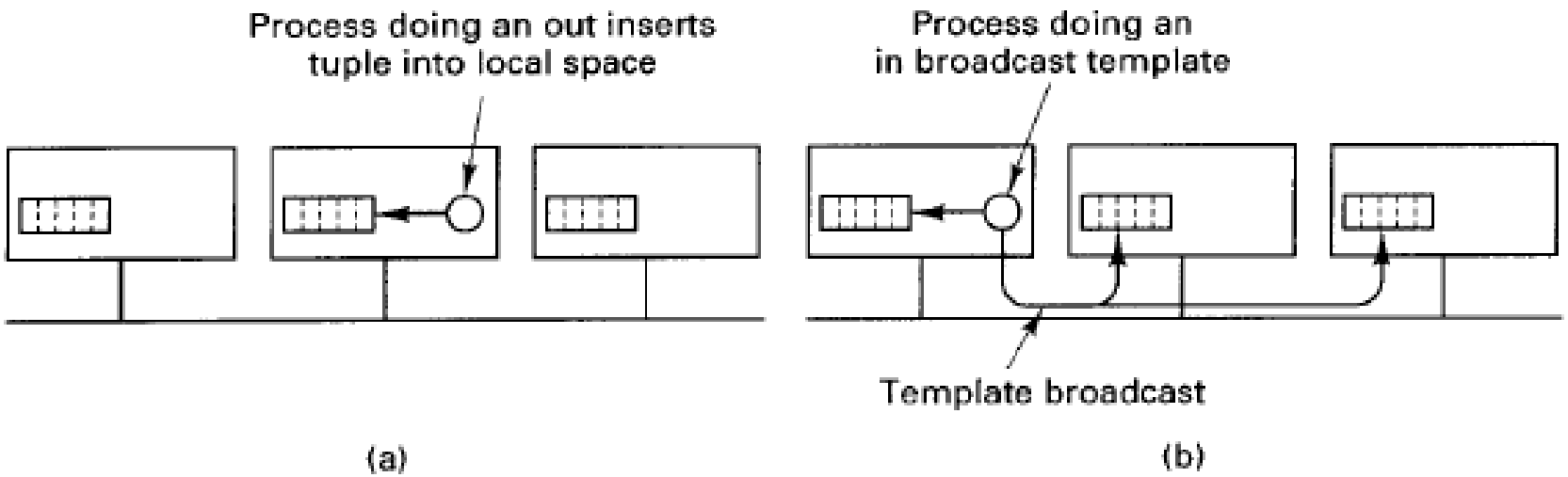


Fig. 6-39. Unreplicated tuple space. (a) An *out* is done locally. (b) An *in* requires the template to be broadcast in order to find a tuple.

Linda (contd…)

3. Partial Replication – It deals with in and out operations.

- When out comes the replication is done on all row machines. Duplicate copies are maintained in row.
- When in comes the replication is done on all column machines, no duplicate copies with in operation.

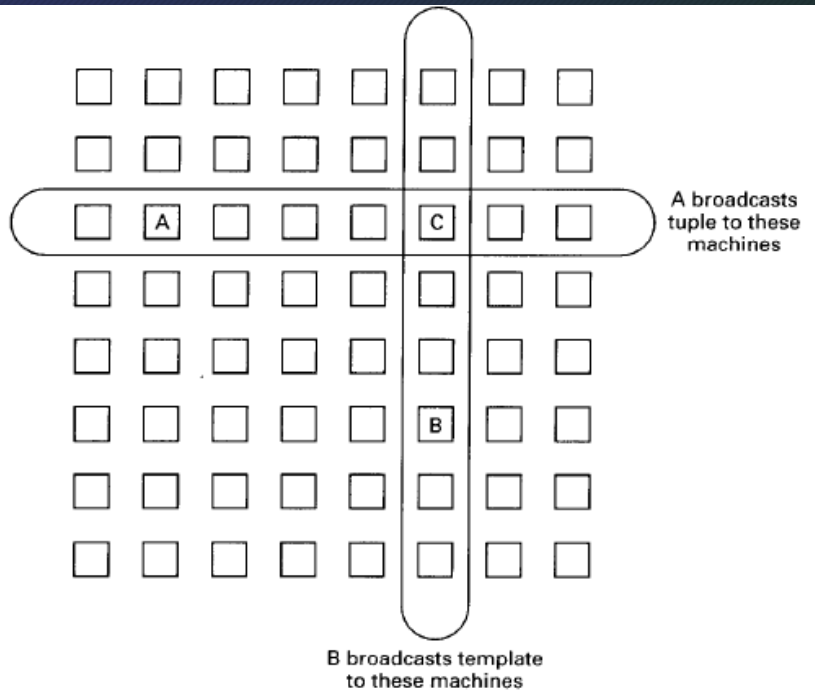


Fig. 6-40. Partial broadcasting of tuples and templates.

ORCA

- Orca is a language and provides runtime environment and it is compiler. Other languages can use runtime environment to get benefits of Orca for other languages. It is a well checked language., checking is done at compile and runtime.
- Two specific interests;
- Objects
- Fork
- Guard till at least one element is there in stack. i.e., pop operation requires at least one element in stack.
- Suspend or block until condition becomes true.

```
Object implementation stack;
    top: integer;                                   # variable indicating the top
    stack: array [integer 0..N–1] of integer;       # storage for the stack
    operation push (item: integer);                 # function returning nothing
    begin
        stack [top] := item;                        # push item onto the stack
        top := top + 1;                             # increment the stack pointer
    end;
    operation pop(): integer;                       # function returning an integer
    begin
        guard top > 0 do                            # suspend if the stack is empty
            top := top – 1;                         # decrement the stack pointer
            return stack [top];                     # return the top item
        od;
    end;
begin
    top := 0;                                       # initialization
end;
```

Fig. 6-41. A simplified stack object, with internal data and two operations.

ORCA

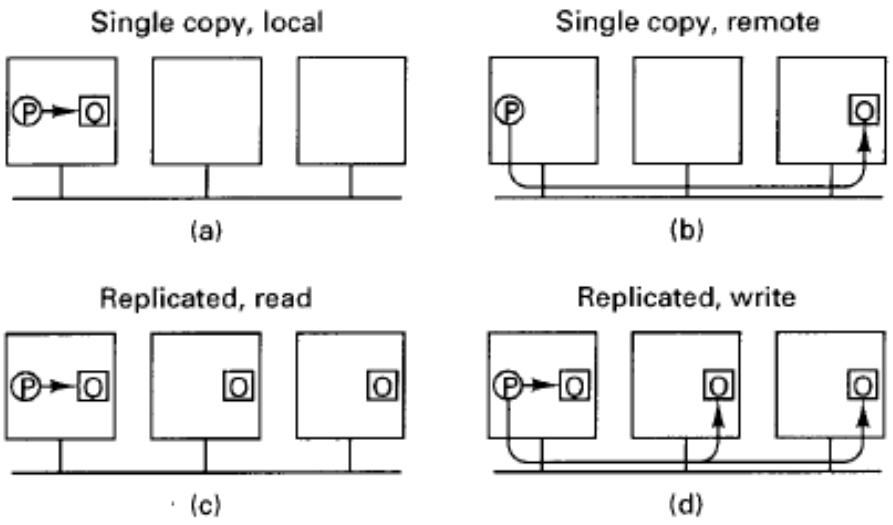


Fig. 6-42. Four cases of performing an operation on an object, *O*.

Fig (a) : single copy on local machine; reading /writing no problem, since it will not effect other copies on other machines.

Fig (b) : Single copy but available on remote machine; solution to RPC. Reading/writing can be done.

Fig(c) : Multiple copies are there on many machines- then reading is no problem; writing/updating has to be informed to all machines; synchronize with other machines for writing. This is only for reading.

Fig(d) : This is only for writing. Multiple copies can exist.

Sequencer is a mechanism or a technique or process used to deliver packets/data reliably on unreliable channel.

THANK YOU

Reference: Distributed Operating Systems by Andrew S. Tanenbaum,
Online Contents