UNIT-5: Distributed Shared Memory

UNIT-5: Distributed Shared Memory

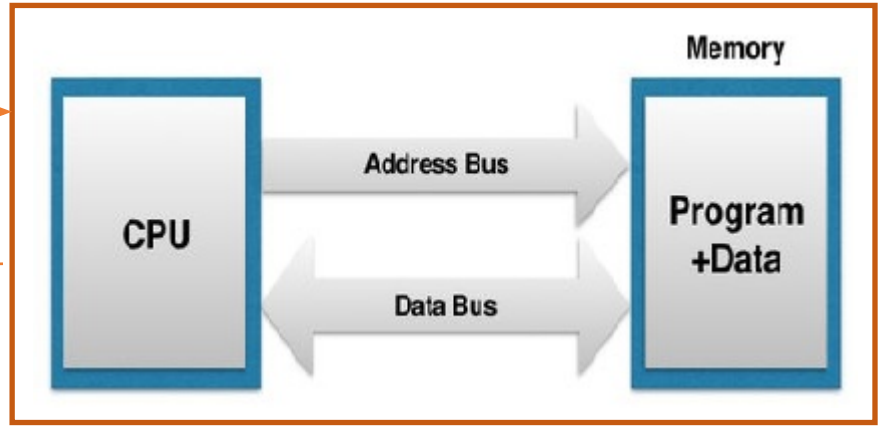
- **Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor**
- Switched Multiprocessors, Directories, Caching
- Protocols – Dash Protocols, NUMA Multiprocessors, NUMA Algorithms
- Consistency Models – Strict, Sequential, Causal, PRAM, Processor, Weak, Release, & Entry Consistency
- Page – Based Distributed Shared Memory
- Shared-Variable Distributed Shared Memory
- Object-Based Distributed Shared Memory

Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor

- On-Chip Memory

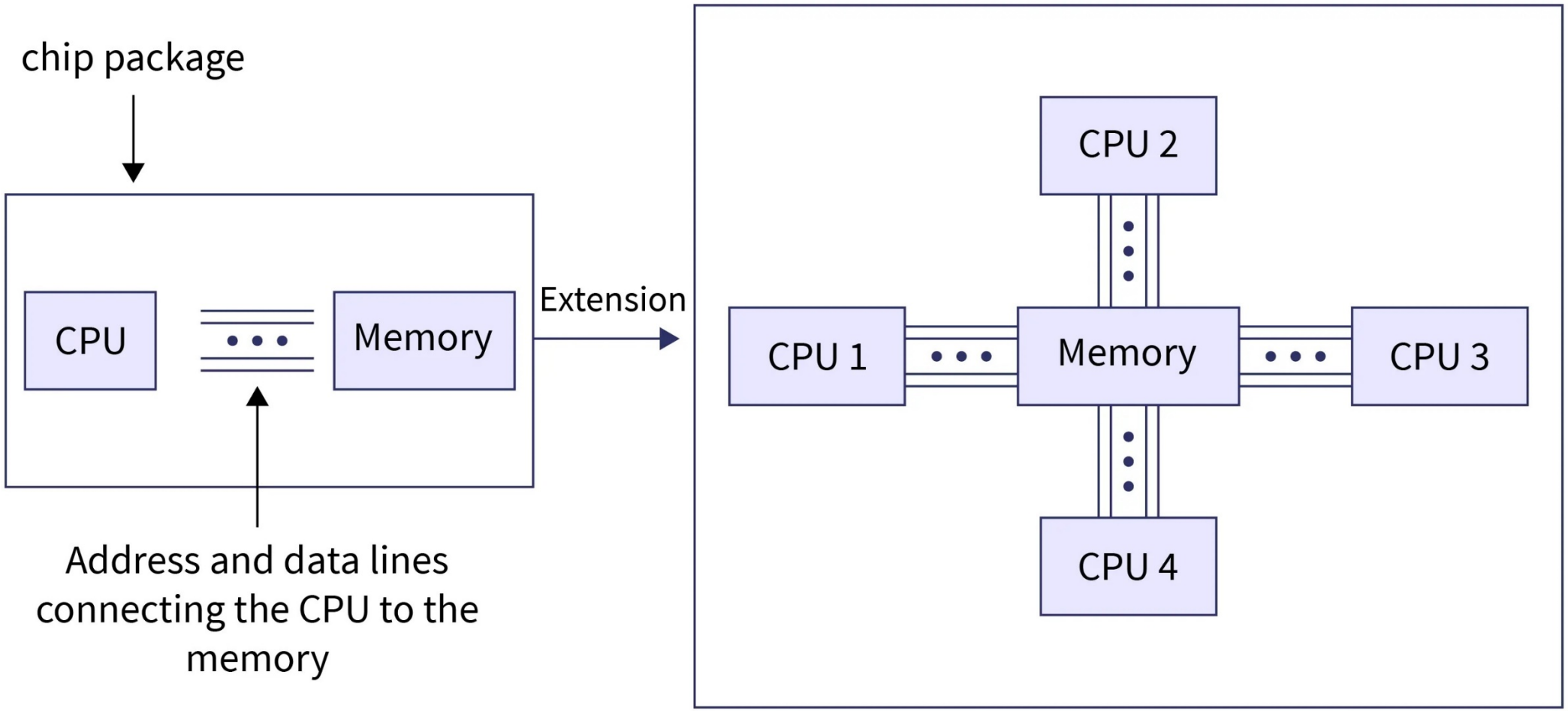
Self-contained chips containing a CPU and all the memory also exist. Such chips are produced by the millions, and are widely used in cars , appliances and even toys. The CPU portion of the chip has address and data lines that directly connect to the memory portion.

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.

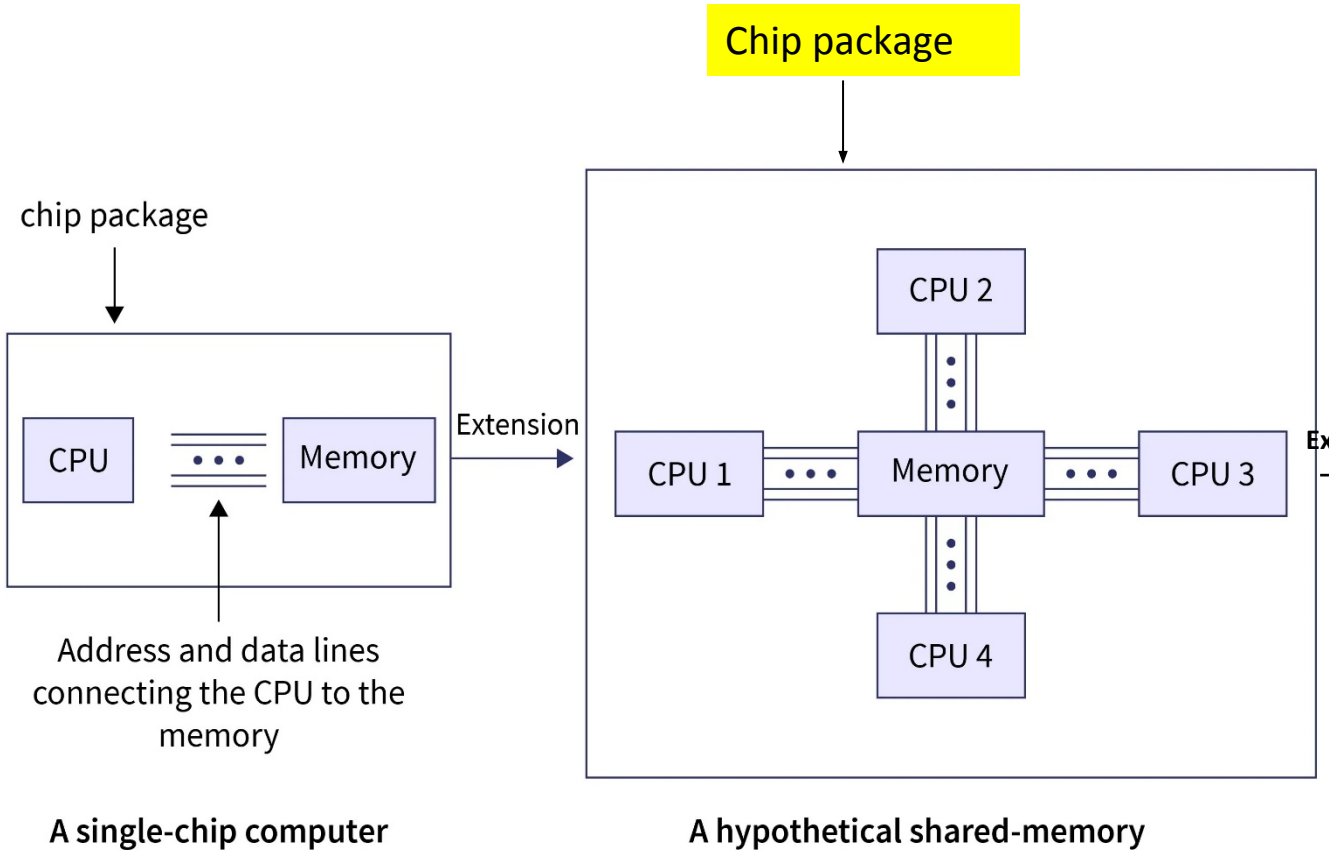
Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor

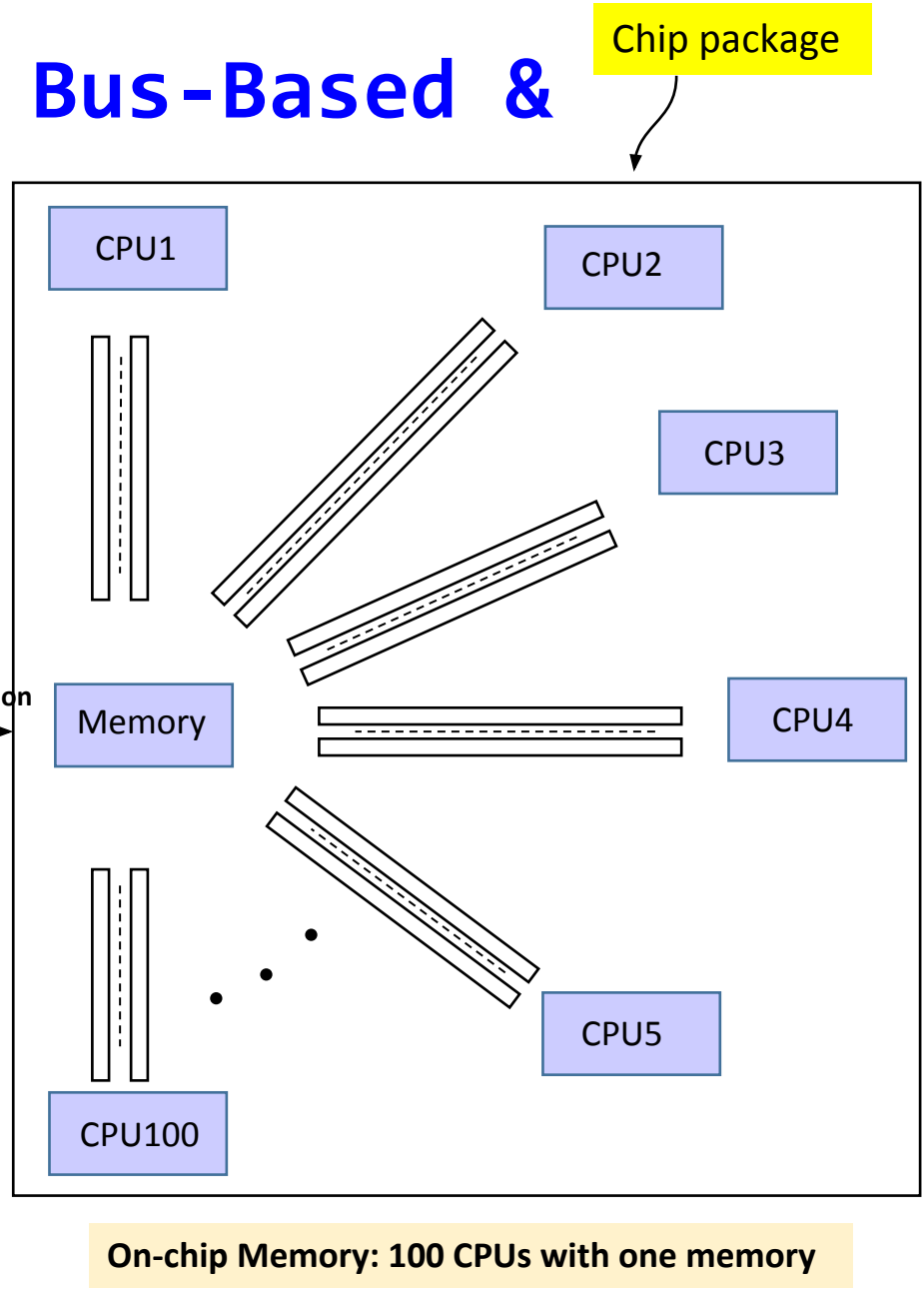


A single-chip computer

A hypothetical shared-memory multiprocessor

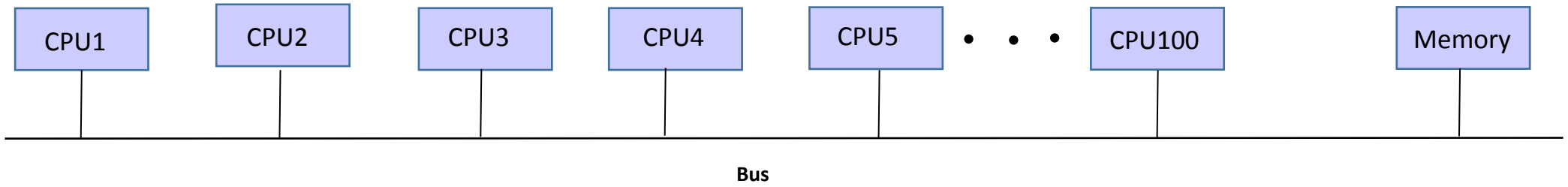
Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor

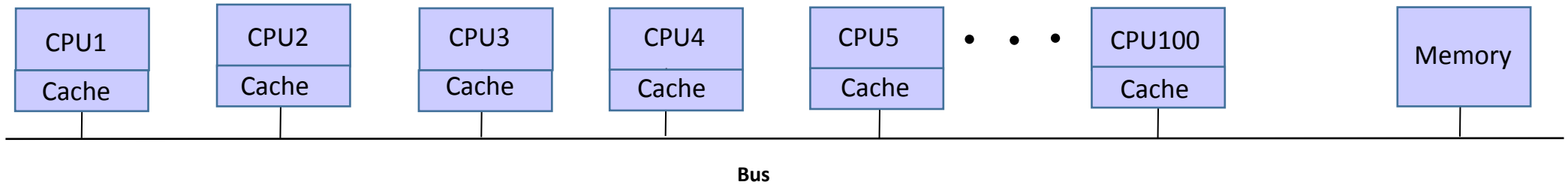




On-chip Memory: 100 CPUs with one memory

Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor





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.

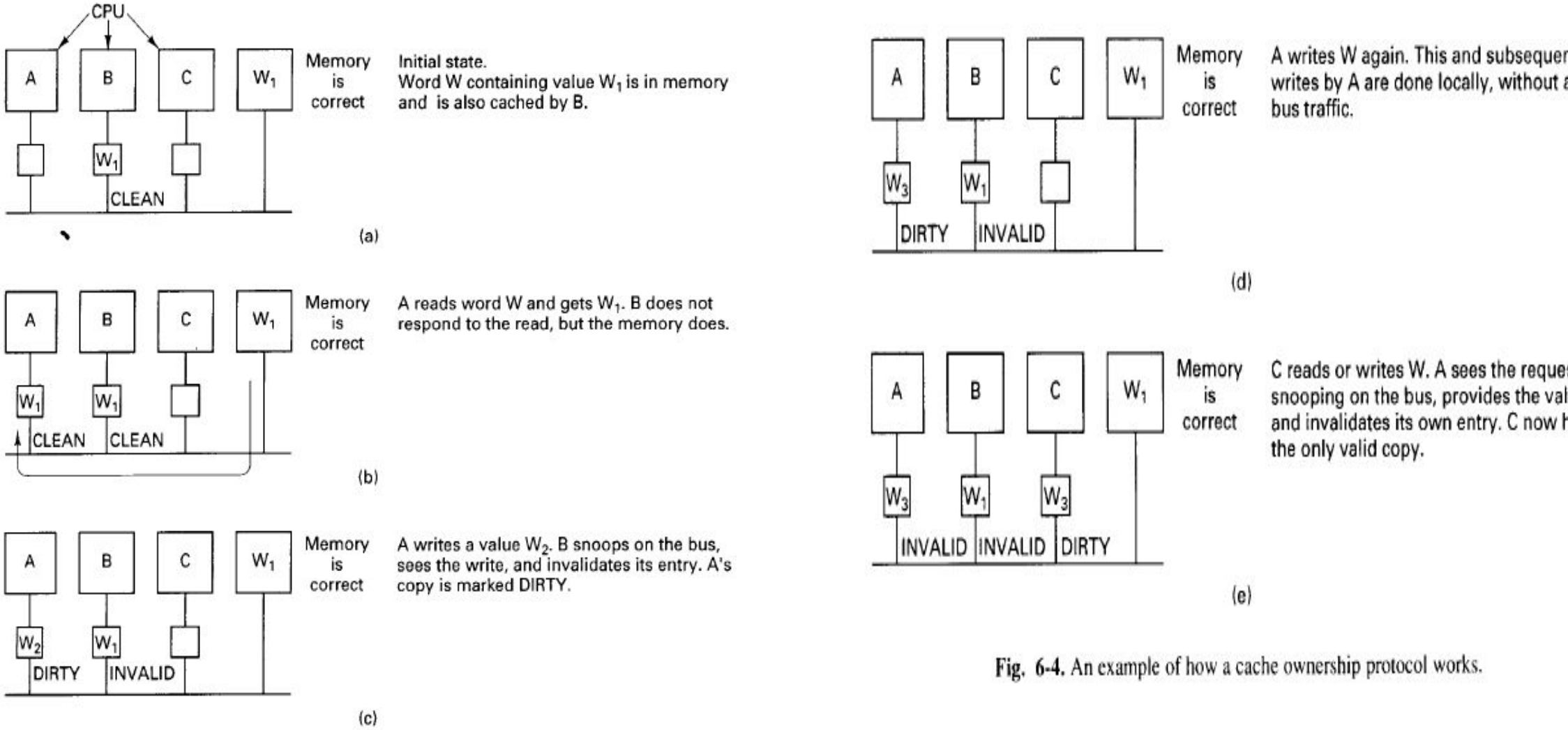
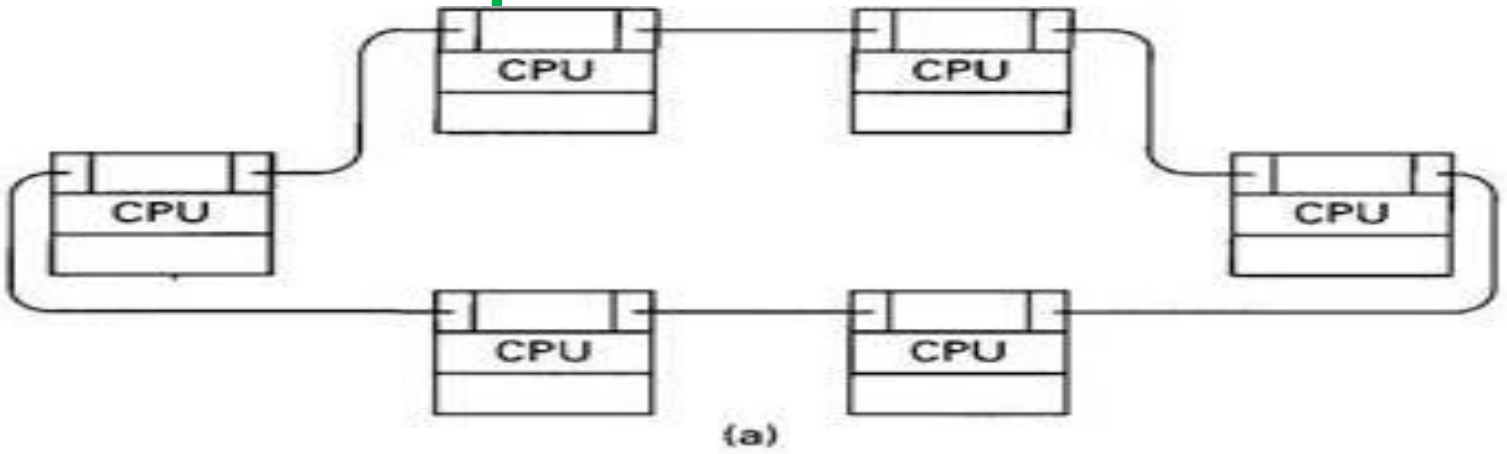
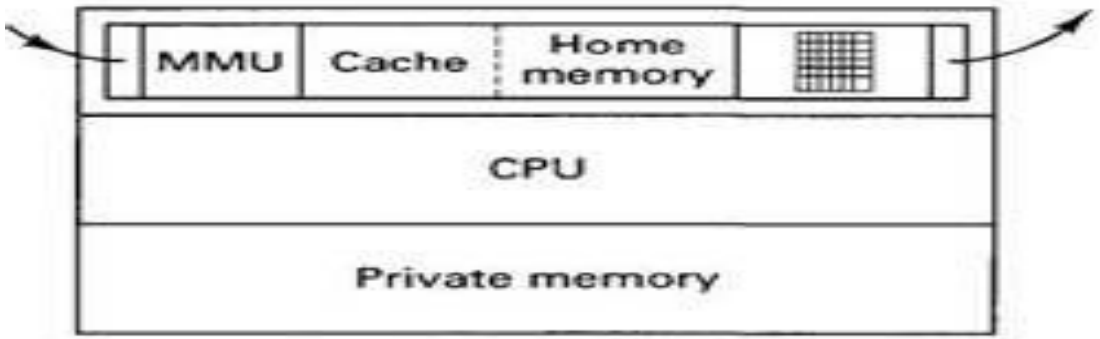


Fig. 6-4. An example of how a cache ownership protocol works.

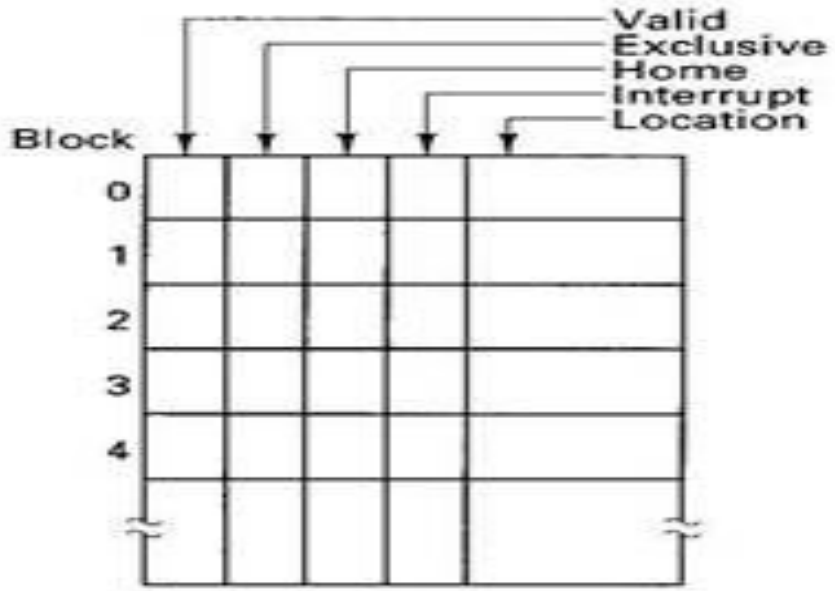
Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor



(a)



(b)



(c)

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

Ring-Based Multiprocessors

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.

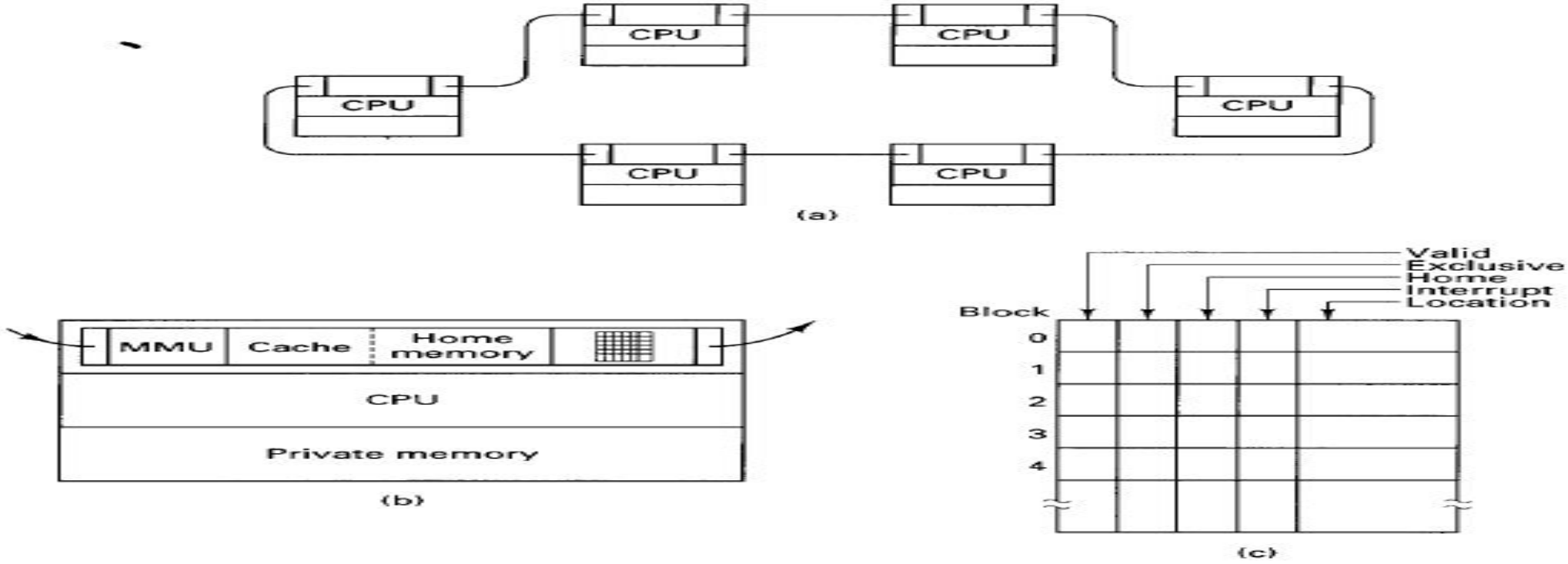


Fig. 6-5. (a) The Memnet ring. (b) A single machine. (c) The block table.

UNIT-5: Distributed Shared Memory

- Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor
- **Switched Multiprocessors, Directories, Caching**
- Protocols – Dash Protocols, NUMA Multiprocessors, NUMA Algorithms
- Consistency Models – Strict, Sequential, Causal, PRAM, Processor, Weak, Release, & Entry Consistency
- Page – Based Distributed Shared Memory
- Shared-Variable Distributed Shared Memory
- Object-Based Distributed Shared Memory

Switched Multiprocessors

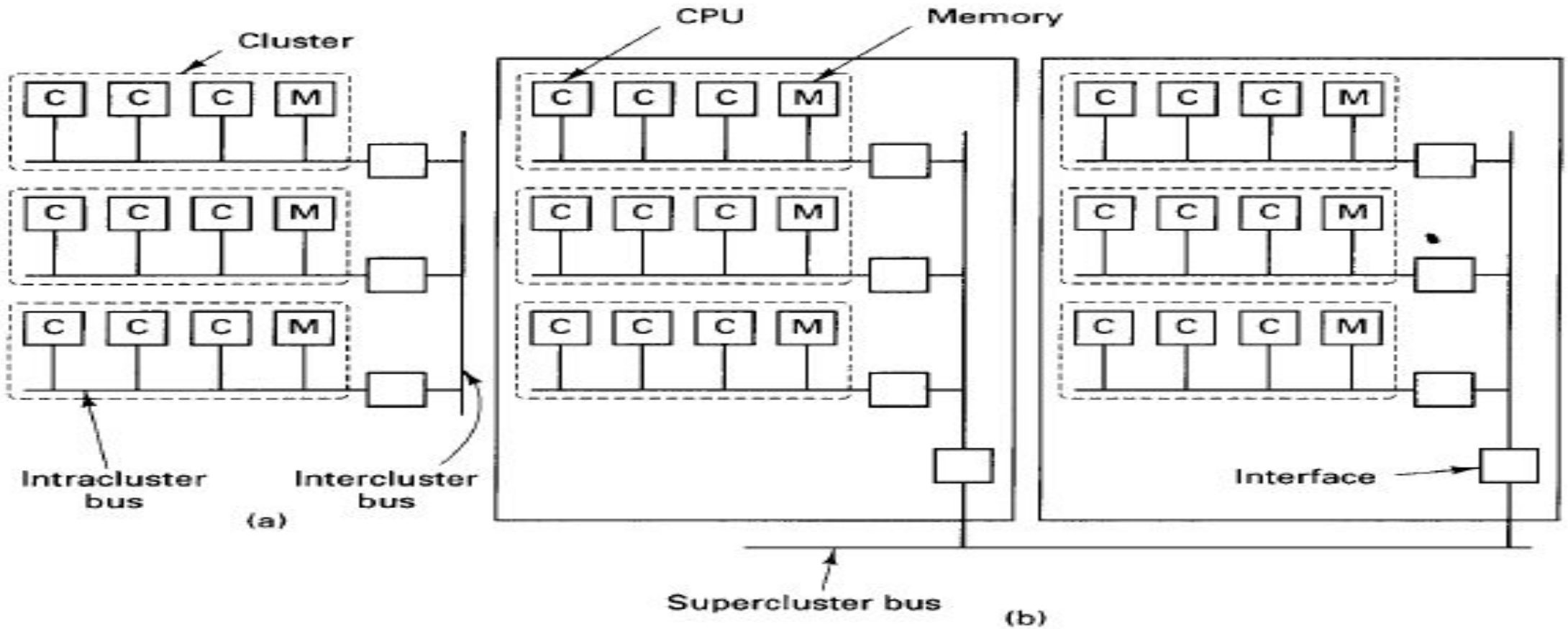
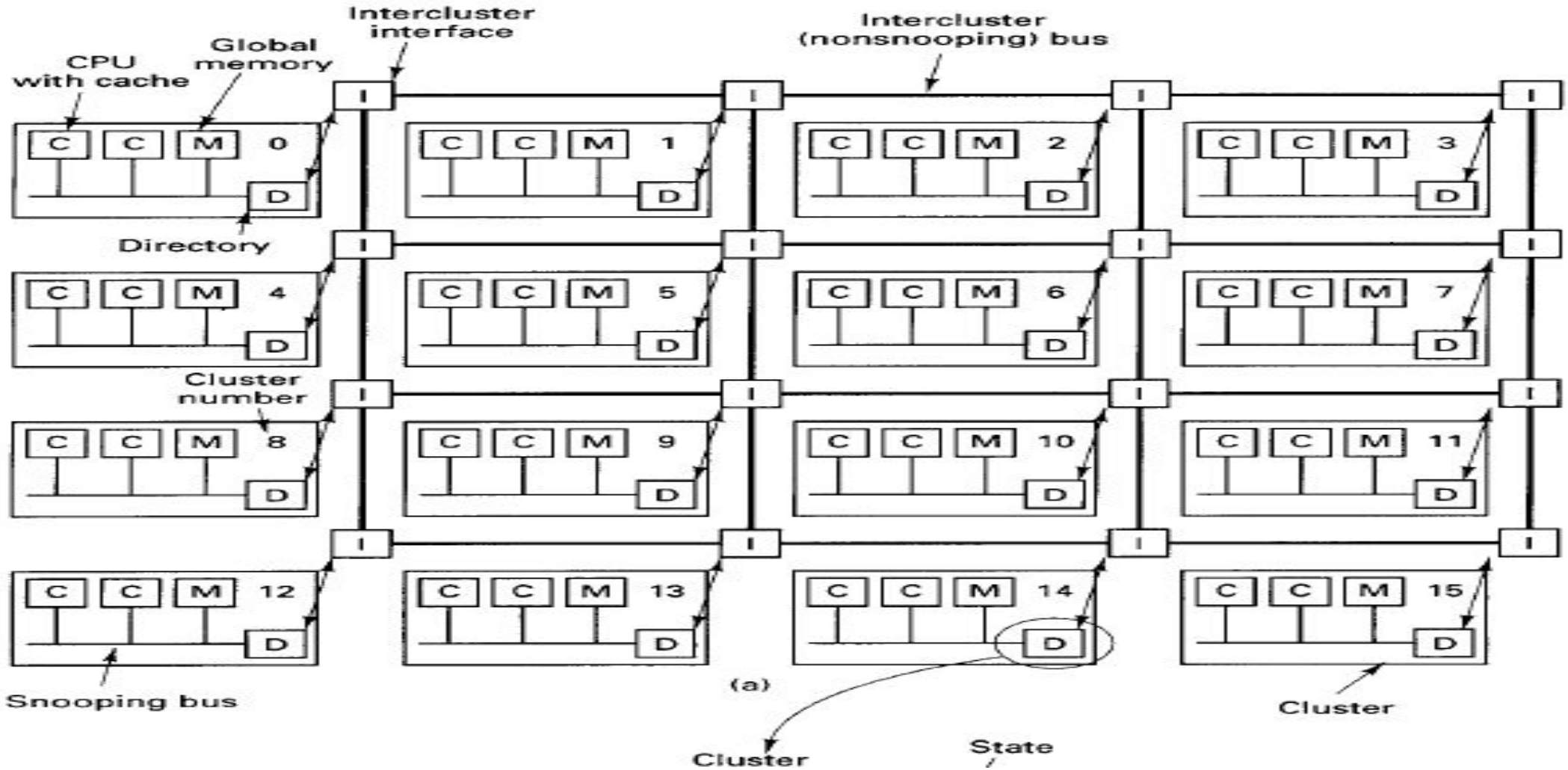


Fig. 6-6. (a) Three clusters connected by an intercluster bus to form one supercluster. (b) Two superclusters connected by a supercluster bus.

Directories



Directories

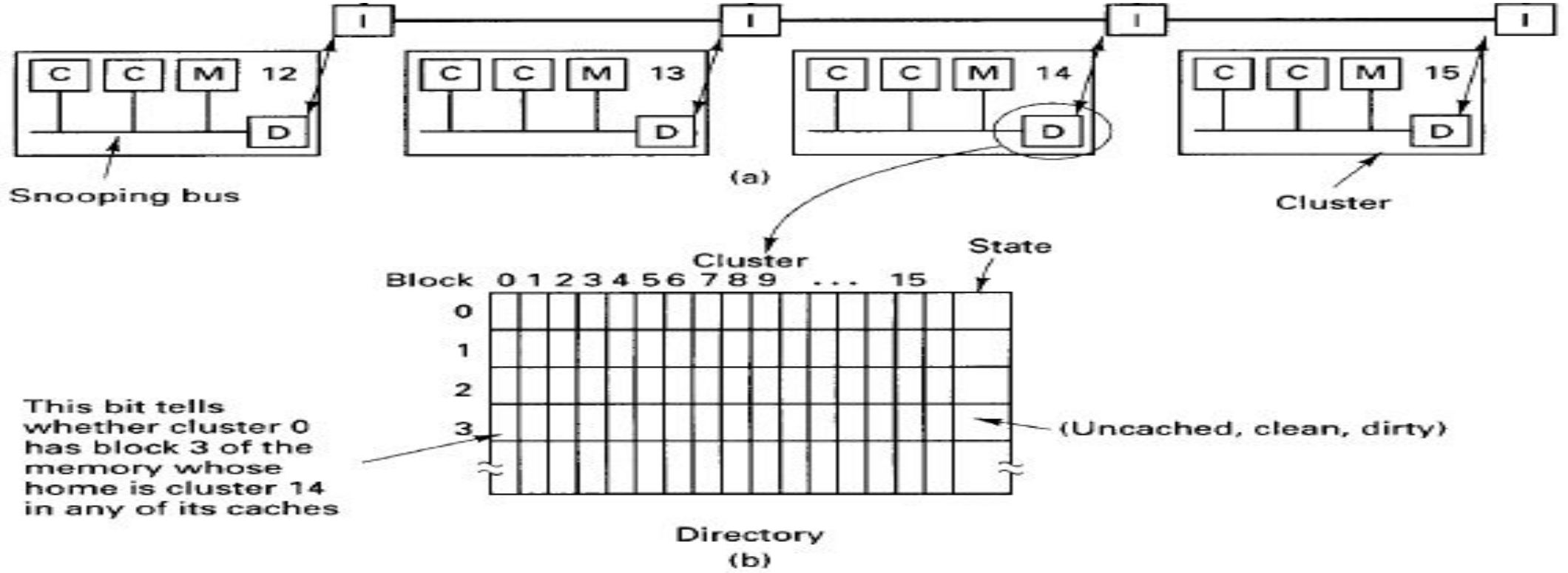


Fig. 6-7. (a) A simplified view of the Dash architecture. Each cluster actually has four CPUs, but only two are shown here. (b) A Dash directory.

Caching

UNCACHED – The only copy of the block is in this memory.

CLEAN – Memory is up-to-date; the block may be in several caches.

DIRTY – Memory is incorrect; only one cache holds the block.

UNIT-5: Distributed Shared Memory

- Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor
- Switched Multiprocessors, Directories, Caching
- **Protocols – Dash Protocols, NUMA Multiprocessors, NUMA Algorithms**
- Consistency Models – Strict, Sequential, Causal, PRAM, Processor, Weak, Release, & Entry Consistency
- Page – Based Distributed Shared Memory
- Shared-Variable Distributed Shared Memory
- Object-Based Distributed Shared Memory

Protocols

Location where the block was found

Block state	R's cache	Neighbor's cache	Home cluster's memory	Some cluster's cache
UNCACHED			Send block to R; mark as CLEAN and cached only in R's cluster	
CLEAN	Use block	Copy block to R's cache	Copy block from memory to R; mark as also cached in R's cluster	
DIRTY	Use block	Send block to R and to home cluster; tell home to mark it as CLEAN and cached in R's cluster		Send block to R and to home cluster (if cached elsewhere); tell home to mark it as CLEAN and also cached in R's cluster

(a)

Location where the block was found

Block state	R's cache	Neighbor's cache	Home cluster's memory	Some cluster's cache
UNCACHED			Send block to R; mark as DIRTY and cached only in R's cluster	
CLEAN	Send message to home asking for exclusive ownership in DIRTY state; if granted, use block	Copy and invalidate block; send message to home asking for exclusive ownership in DIRTY state	Send block to R; invalidate all cached copies; mark it as DIRTY and cached only in R's cluster	
DIRTY	Use block	Cache-to-cache transfer to R; invalidate neighbor's copy		Send block directly to R; invalidate cached copy; home marks it as DIRTY and cached only in R's cluster

(b)

Fig. 6-8. Dash protocols. The columns show where the block was found. The rows show the state it was in. The contents of the boxes show the action taken. *R* refers to the requesting CPU. An empty box indicates an impossible situation. (a) Reads. (b) Writes.

UNIT-5: Distributed Shared Memory

- Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor
- Switched Multiprocessors, Directories, Caching
- Protocols – Dash Protocols, NUMA Multiprocessors, NUMA Algorithms
- **Consistency Models – Strict, Sequential, Causal, PRAM, Processor, Weak, Release, & Entry Consistency**
- Page – Based Distributed Shared Memory
- Shared-Variable Distributed Shared Memory
- Object-Based Distributed Shared Memory

Consistency Models

Strict Consistency

Any read to a memory location x returns the value stored by the most recent write operation to x.

Sequential Consistency

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.

Causal Consistency

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

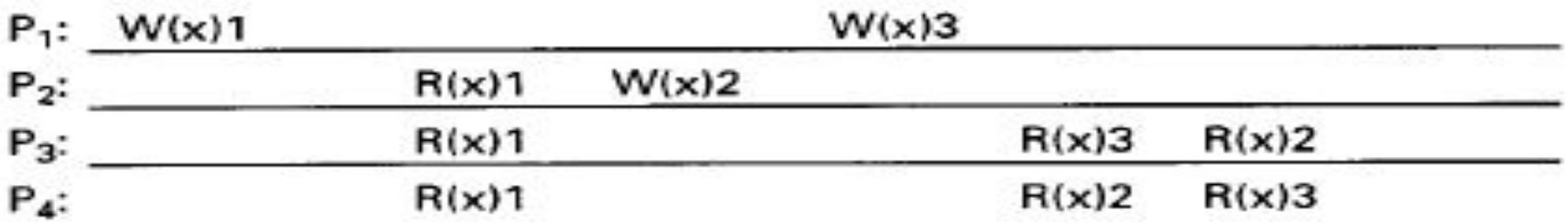


Fig. 6-16. This sequence is allowed with causally consistent memory, but not with sequentially consistent memory or strictly consistent memory.

PRAM Consistency

Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

Processor Consistency

For every memory location, x, there be global agreement about the order of writes to x. Writes to different locations need not be viewed in the same order by different processes.

Weak Consistency

Weak Consistency has three properties:

1. Accesses to synchronization variables are sequentially consistent.
2. No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
3. No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.

Release Consistency

Release consistency obeys the following rules:

1. Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.
2. Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.
3. The acquire and release accesses must be processor consistent (sequential consistency is not required).

Entry Consistency

Entry consistency meets all the following conditions.

1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
3. After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters
Sequential	All processes see all shared accesses in the same order
Causal	All processes see all casually-related shared accesses in the same order
Processor	PRAM consistency + memory coherence
PRAM	All processes see writes from each processor in the order they were issued. Writes from different processors may not always be seen in the same order

(a)

Weak	Shared data can only be counted on to be consistent 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)

Fig. 6-24. (a) Consistency models not using synchronization operations. (b) Models with synchronization operations.

UNIT-5: Distributed Shared Memory

- Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor
- Switched Multiprocessors, Directories, Caching
- Protocols – Dash Protocols, NUMA Multiprocessors, NUMA Algorithms
- Consistency Models – Strict, Sequential, Causal, PRAM, Processor, Weak, Release, & Entry Consistency
- **Page – Based Distributed Shared Memory**
- Shared-Variable Distributed Shared Memory
- Object-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

1. Access remotely – no copying page into local memory, it is needed for temporary usage.
2. Fetch – copying actual page into local memory.

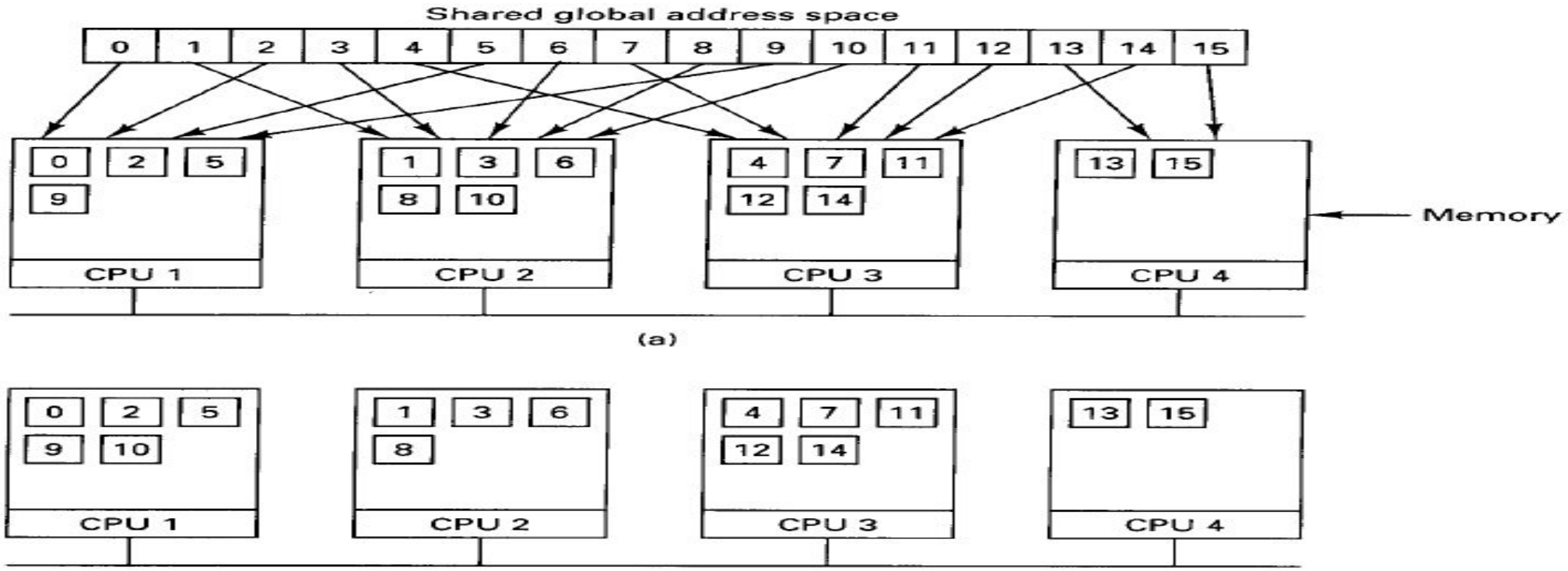
NORMA

1. No remote access.
2. 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.

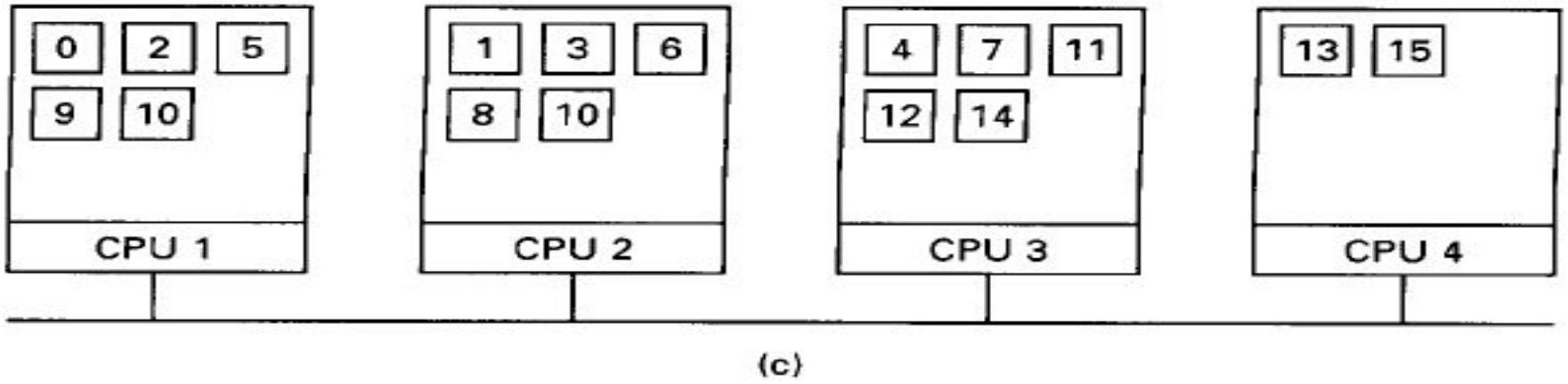


Fig. 6-25. (a) Chunks of address space distributed among four machines. (b) Situation after CPU 1 references chunk 10. (c) Situation if chunk 10 is read only and replication is used.

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.

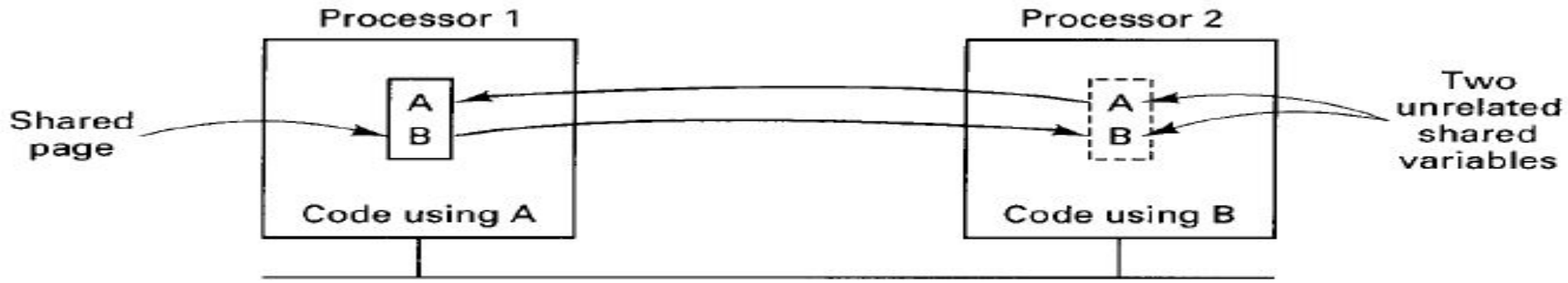
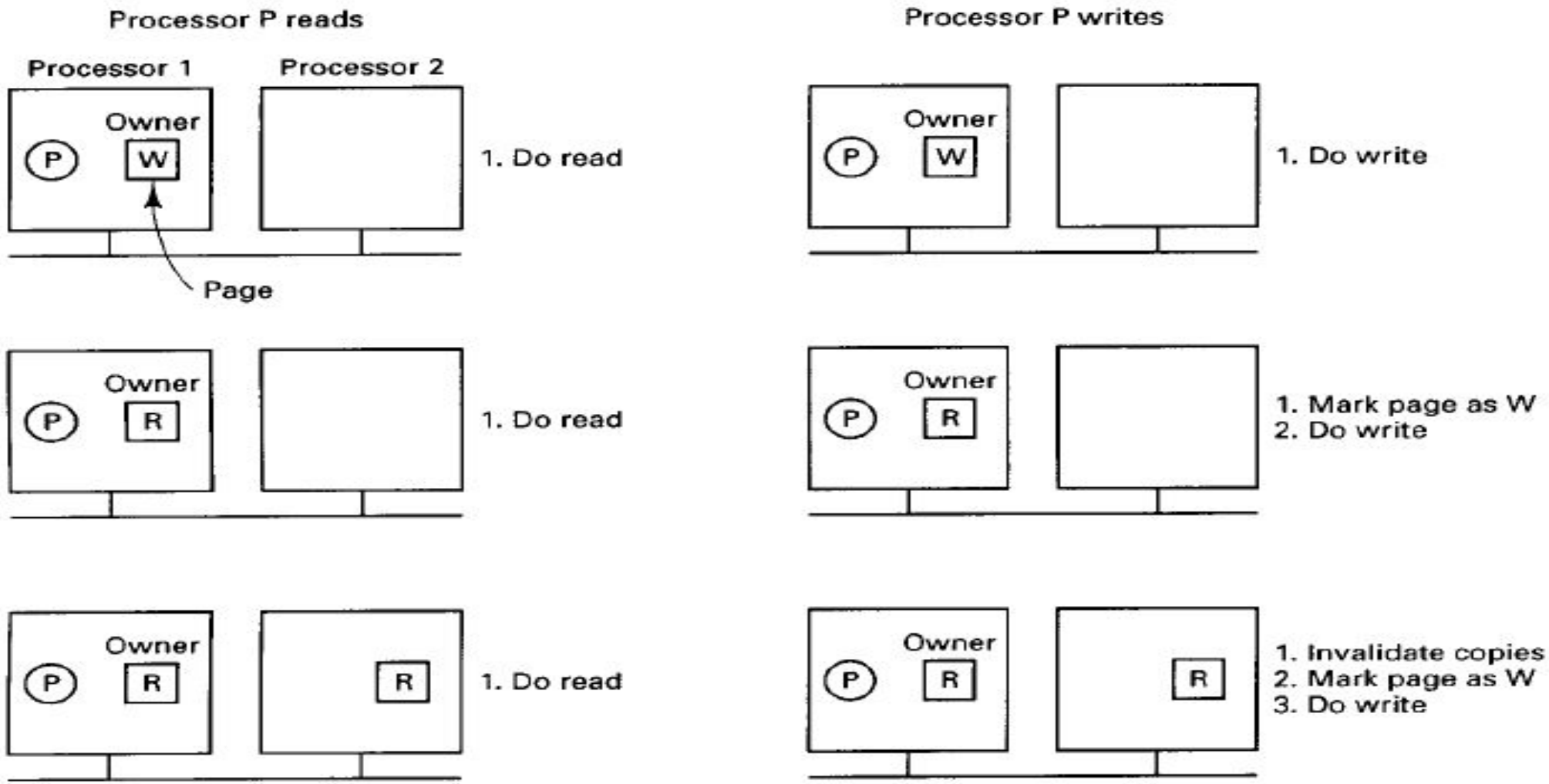


Fig. 6-26. False sharing of a page containing two unrelated variables.

Achieving Sequential Consistency

There is only one write copy at the maximum or sometimes zero.



Achieving Sequential Consistency

There is only one write copy at the maximum or sometimes zero.

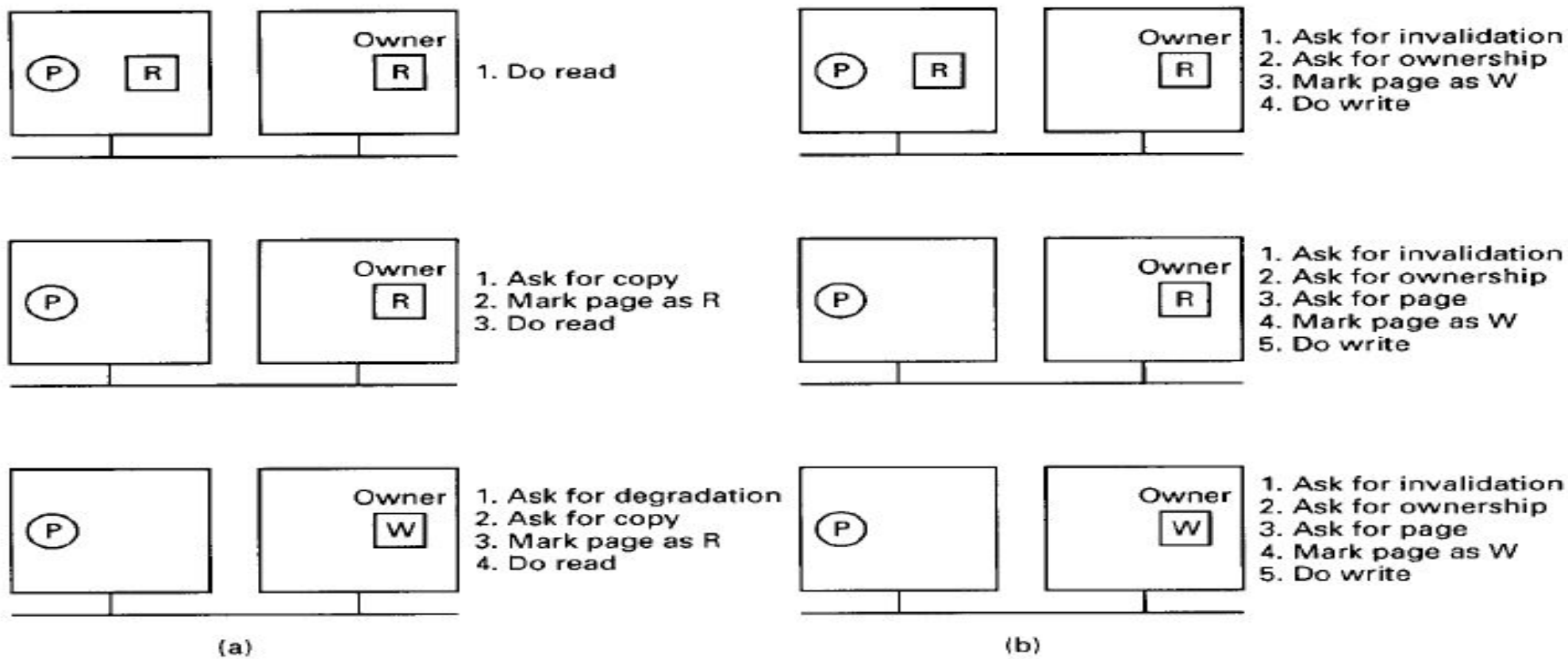


Fig. 6-27. (a) Process *P* wants to read a page. (b) Process *P* wants to write a page.

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.

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).

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.

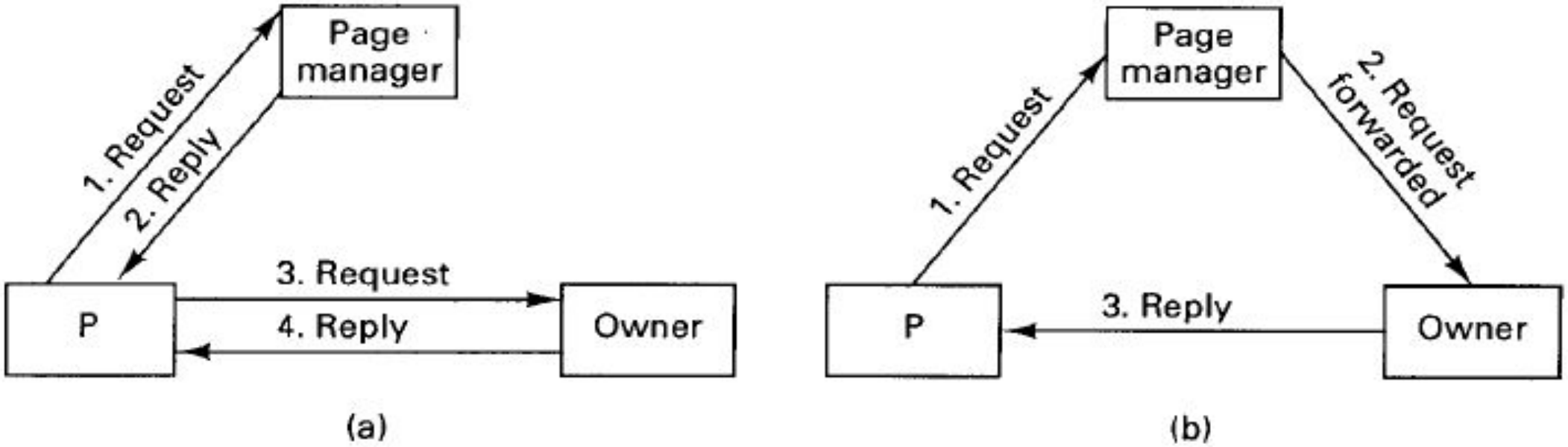


Fig. 6-28. Ownership location using a central manager. (a) Four-message protocol. (b) Three-message protocol.

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

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.

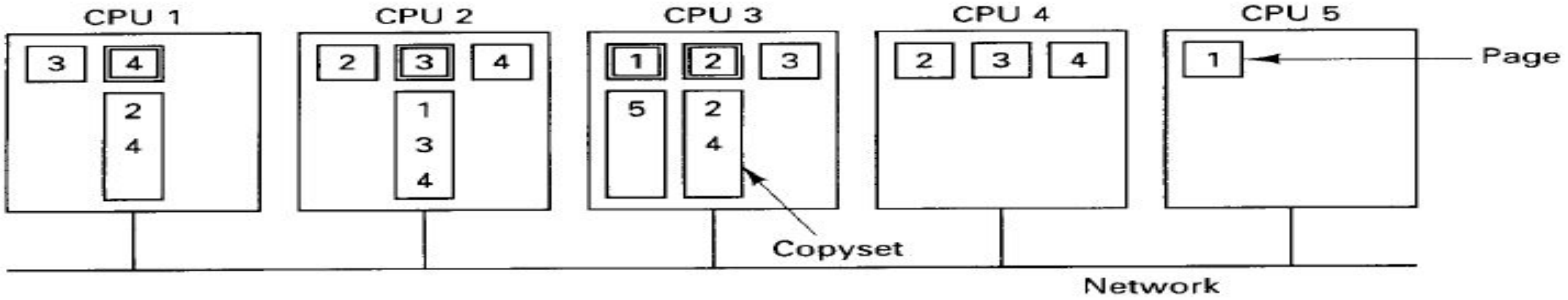


Fig. 6-29. The owner of each page maintains a copyset telling which other CPUs are sharing that page. Page ownership is indicated by the double boxes.

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:

1. First priority is case (3); i.e., drop the pages for whom you are not owner.
2. 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).
3. Third priority is case (1); send this page to some other CPU's or store it in harddisk 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.

UNIT-5: Distributed Shared Memory

- Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor
- Switched Multiprocessors, Directories, Caching
- Protocols – Dash Protocols, NUMA Multiprocessors, NUMA Algorithms
- Consistency Models – Strict, Sequential, Causal, PRAM, Processor, Weak, Release, & Entry Consistency
- Page – Based Distributed Shared Memory
- **Shared-Variable Distributed Shared Memory**
- Object-Based 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.

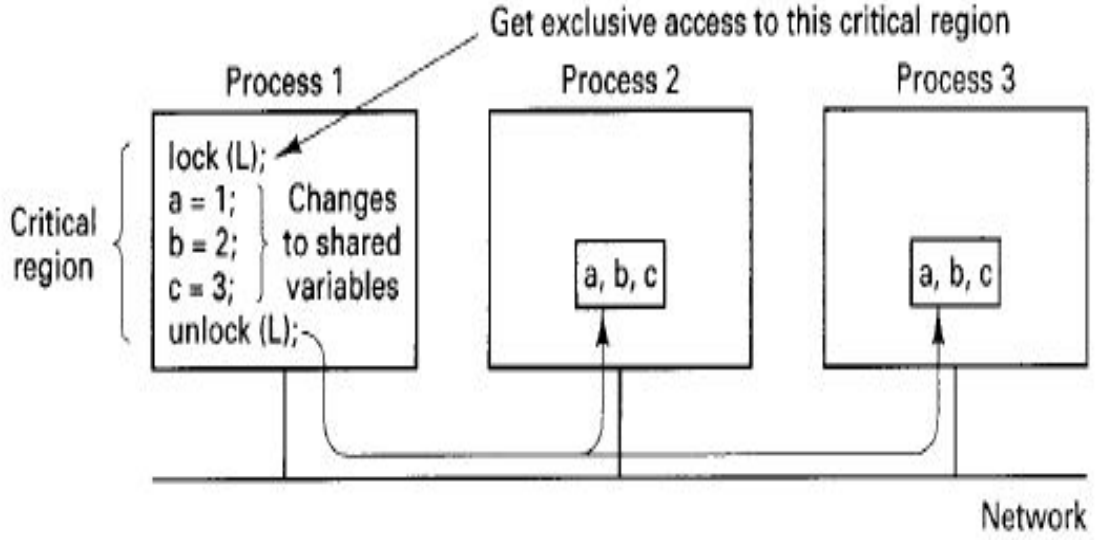


Fig. 6-30. Release consistency in Munin.

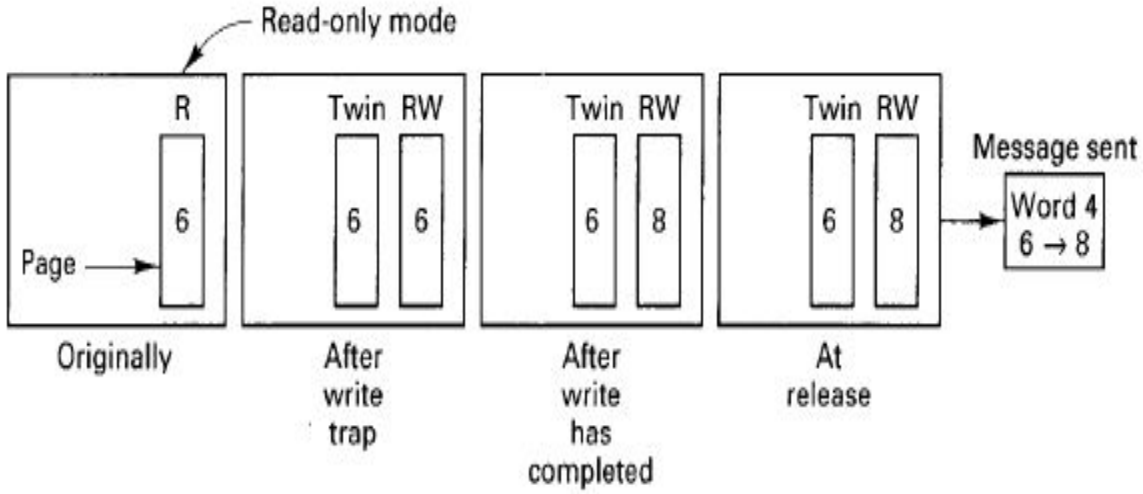


Fig. 6-31. Use of twin pages in Munin.

Multiwriter 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)

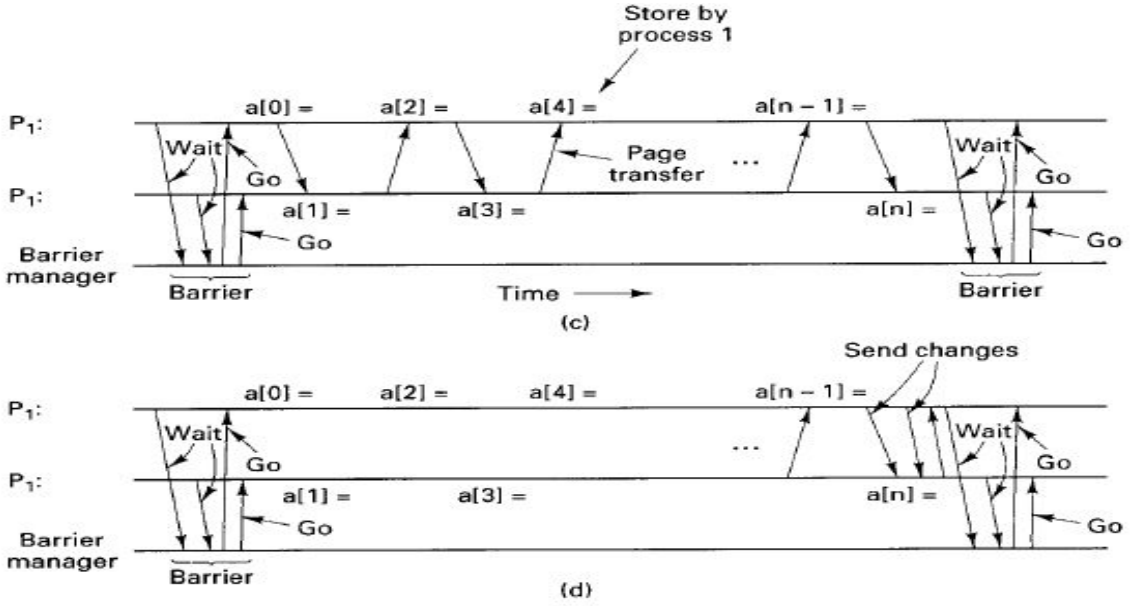


Fig. 6-32. (a) A program using *a*. (b) Another program using *a*. (c) Messages sent for sequentially consistent memory. (d) Messages sent for release consistent memory.

UNIT-5: Distributed Shared Memory

- Architecture – On-Chip Memory, Bus-Based & Ring-Based Multiprocessor
- Switched Multiprocessors, Directories, Caching
- Protocols – Dash Protocols, NUMA Multiprocessors, NUMA Algorithms
- Consistency Models – Strict, Sequential, Causal, PRAM, Processor, Weak, Release, & Entry Consistency
- Page – Based Distributed Shared Memory
- Shared-Variable Distributed Shared 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.

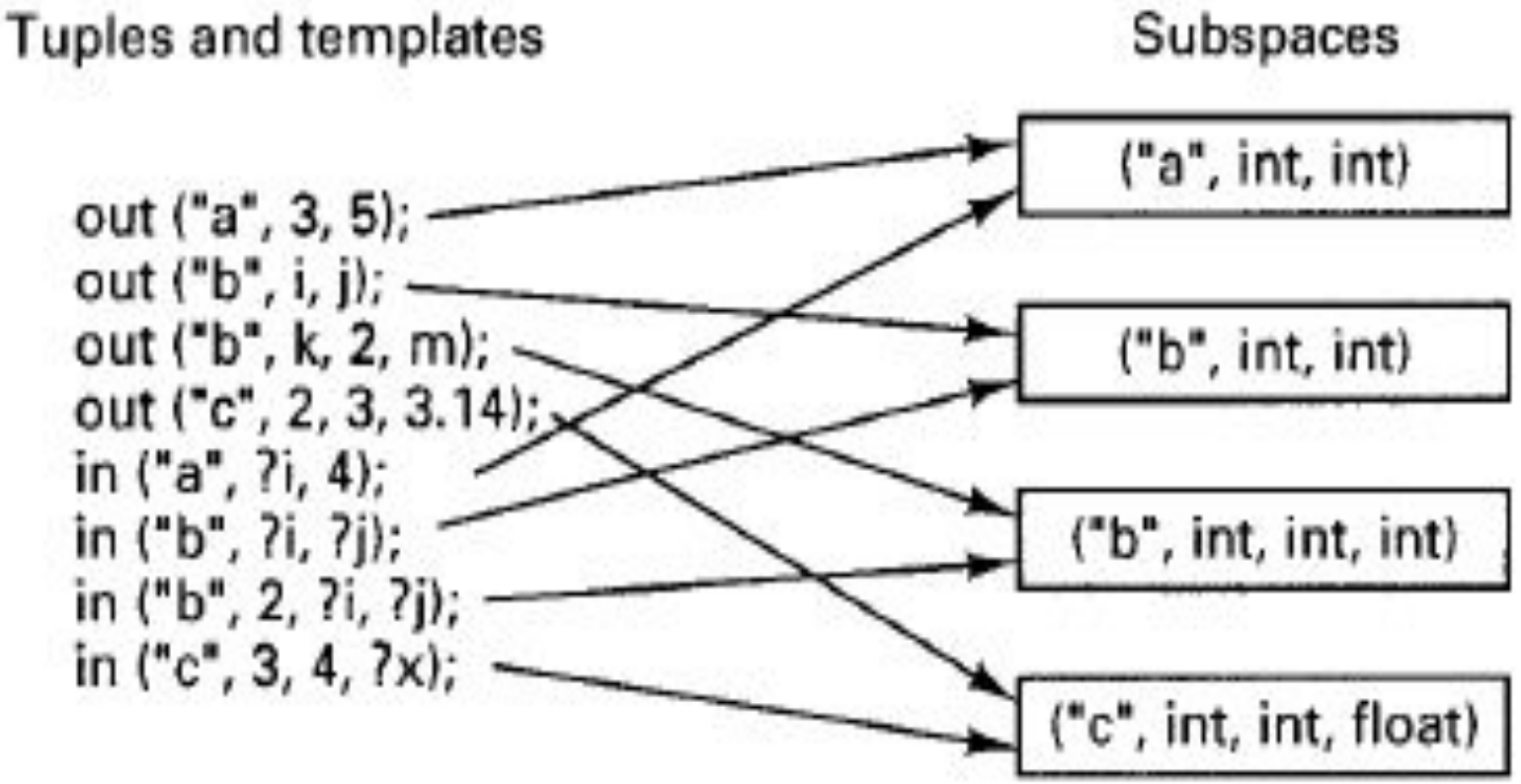


Fig. 6-37. Tuples and templates can be associated with subspaces.

1. Complete replication on each and every machine. Fig 6-38.

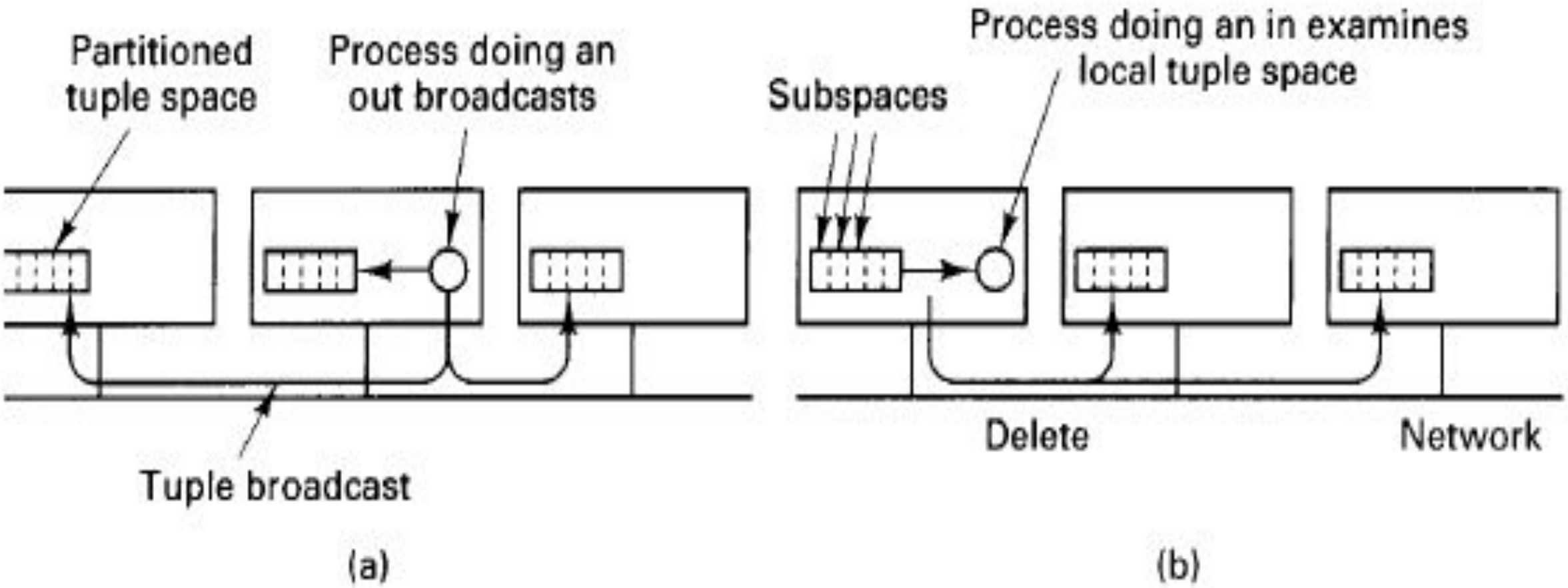


Fig. 6-38. Tuple space can be replicated on all machines. The dotted lines show the partitioning of the tuple space into subspaces. (a) Tuples are broadcast on *out*. (b). *In*s are local, but the deletes must be broadcast.

2. No replication – independent replication of tuple space – deleting from one machine and replicating in another machine. Fig 6-39.

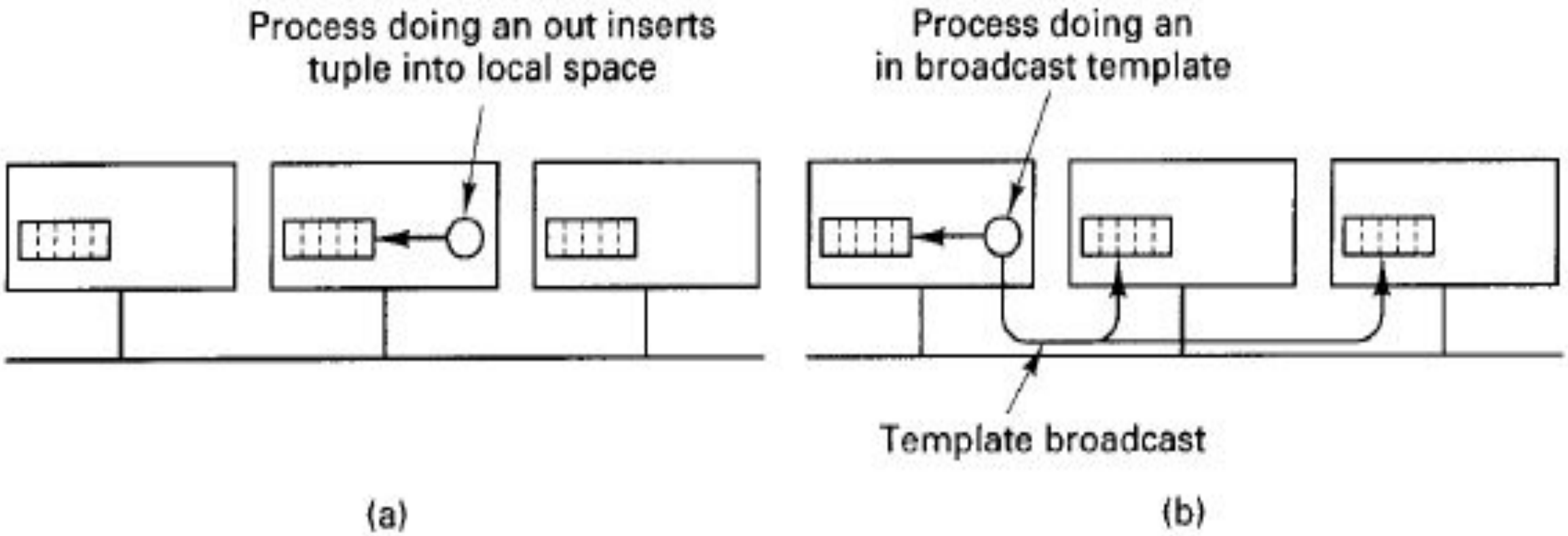


Fig. 6-39. Unreplicated tuple space. (a) An *out* is done locally. (b) An *in* requires the template to be broadcast in order to find a tuple.

3. Partial replication – It deals with in and out operations. Fig 6-40.

When out comes the replication is done on all row machines. Duplicate copies are maintained in row.

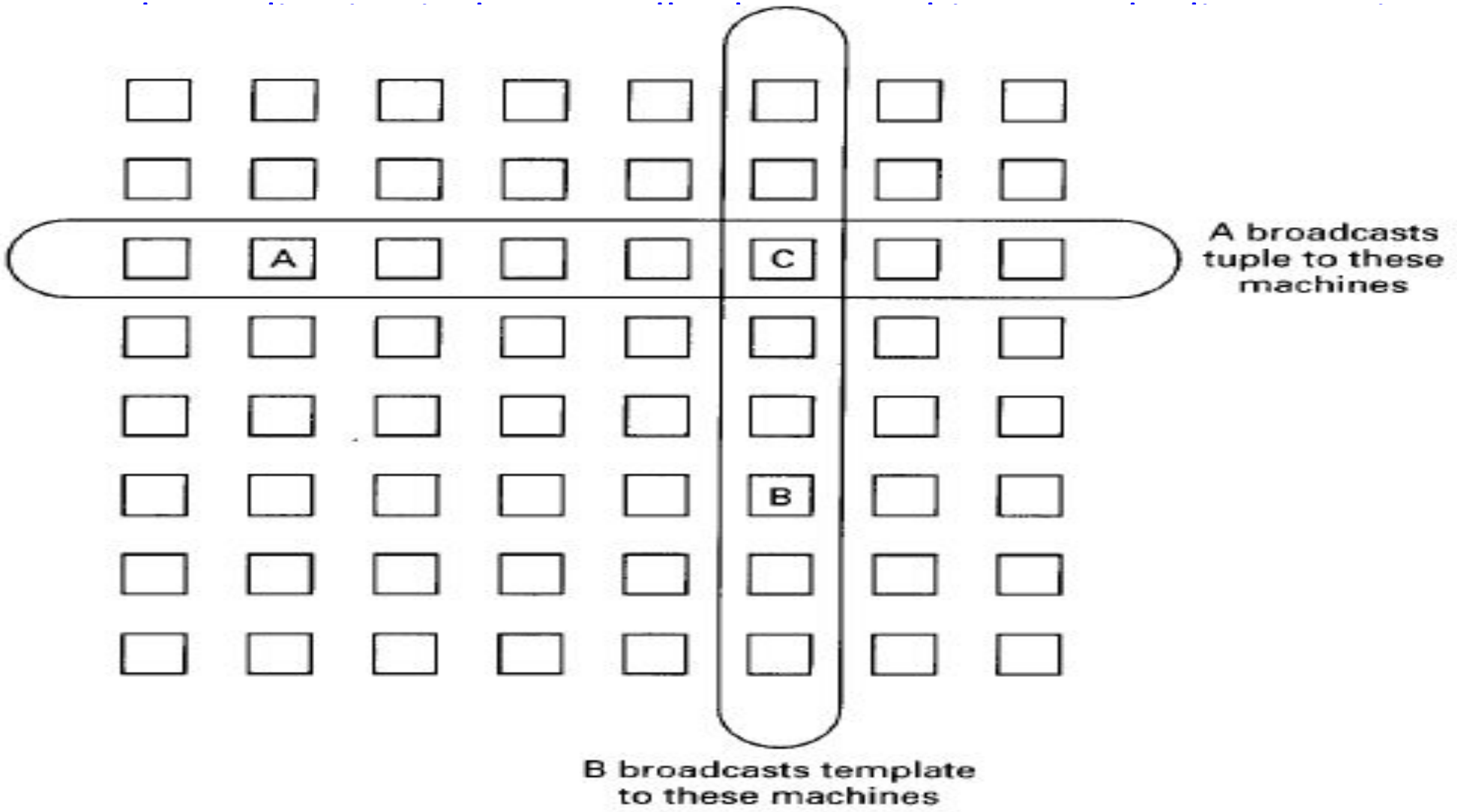


Fig. 6-40. Partial broadcasting of tuples and templates.

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.

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.

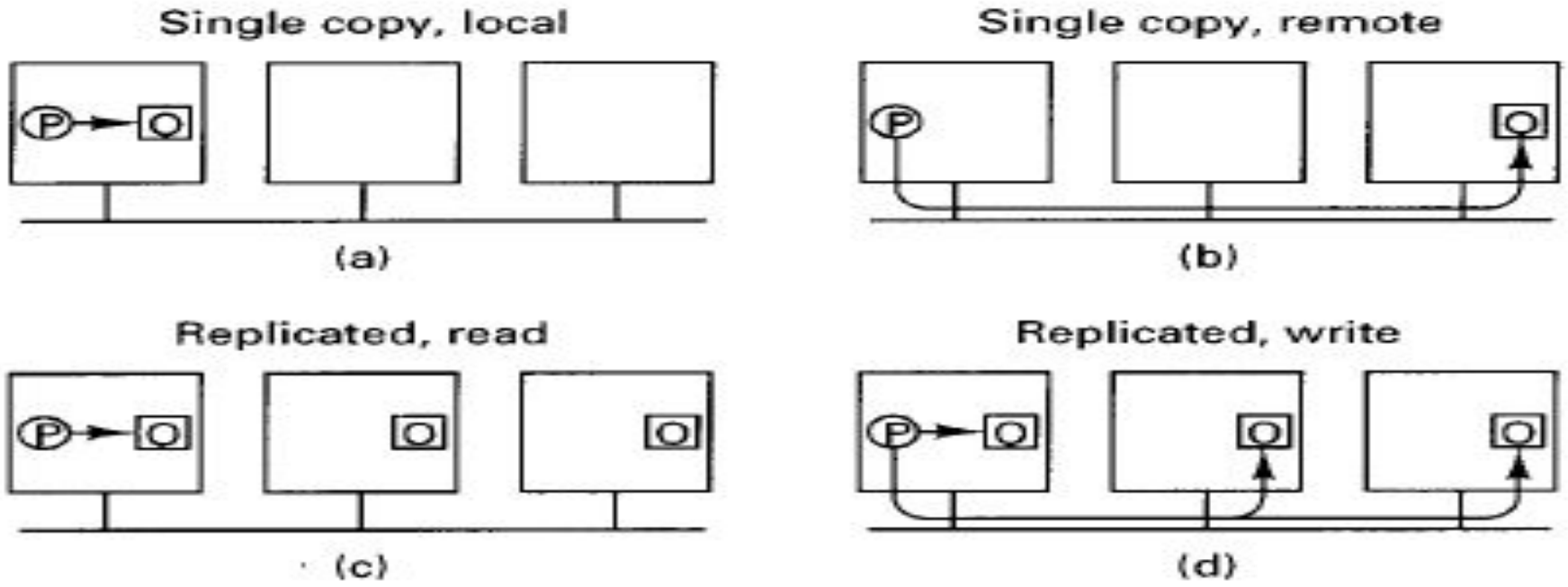


Fig. 6-42. Four cases of performing an operation on an object, *O*.

Q&A

Thank You!