Unit 5-
Distributed Operating Systems

Syllabus

- Distributed Shared memory-Introduction-
- Bus-Based Multiprocessors
- Switched multiprocessors
- Ring Based Multiprocessors
- Numa multiprocessors
- Comparison of shared memory systems
- Consistency models- strict consistency- casual consistency- PRAM consistency
- Weak consistency release consistency- entry consistency
- Page based distributed shared memory-replication-granularity-
- Page replacement- synchronisation
- Shared- variable
- OO distributed shared memory
- DOO architecture- Distributed Object Oriented Process- DOO Communication

Introduction

- Distributed Shared Memory (DSM): (Li-1986, Hudak 1989)
- It is the **collection of workstations connected** by a **LAN** share a **single paged, virtual address space**.
- As a simplest **variant, each page** is present on exactly **one machine**
- A **reference to a local pages** are done in hardware at full memory speed.
- **Page fault** also occurs when a **reference to a page happens on a different machine**.

What happens when page fault occurs:

- The operating system then **sends a message to the remote machine**, which **finds the needed page and sends it to the requesting processor**.
- The **faulting instruction is then restarted** and can now complete.

DSM-Contd..

- In this model, one does not think of each machine as having direct access to an ordinary memory but rather, to a **collection of shared variables**, giving a **higher level of abstraction**.
- Not only does this strategy greatly reduce the amount of data that must be shared, but in most cases, considerable information about the shared data is available, such as their types, which can help optimize the implementation.

DSM –Contd..

- One possible optimization is to **replicate the shared variables on multiple machines**.
- By sharing replicated variables **instead of entire pages**, the **problem of simulating a multiprocessor has been reduced** to that of how to keep multiple copies of a set of typed data structures consistent.
- Potentially, **reads can be done locally without any network traffic, and writes can be done using a multicopy update protocol**.

Objects and Methods

- **Instead of just sharing variables** we could share **encapsulated data types**, often called **objects**.
- These differ from shared variables in that each object has not only some data, but also procedures, called **methods**, that act on the data.
- Programs may only **manipulate an object's data** by **invoking its methods**.
- **Direct access to the data is not permitted**.

Bus–Based Multiprocessors

Bus

- The **connection between** the **CPU** and the **memory** is a **collection of parallel wires, some** holding the **address the CPU wants to read or write**, some for **sending or receiving data**, and the rest for **controlling the transfers**.
- Such a collection of wires is called a **bus**.

How the Bus is placed:

- In most systems, buses are external and are used to connect printed circuit boards containing CPUs, memories, and I/O controllers.
- On a desktop computer, the bus is typically etched onto the main board (the parent-board), which holds the CPU and some of the memory, and into which I/O cards are plugged.

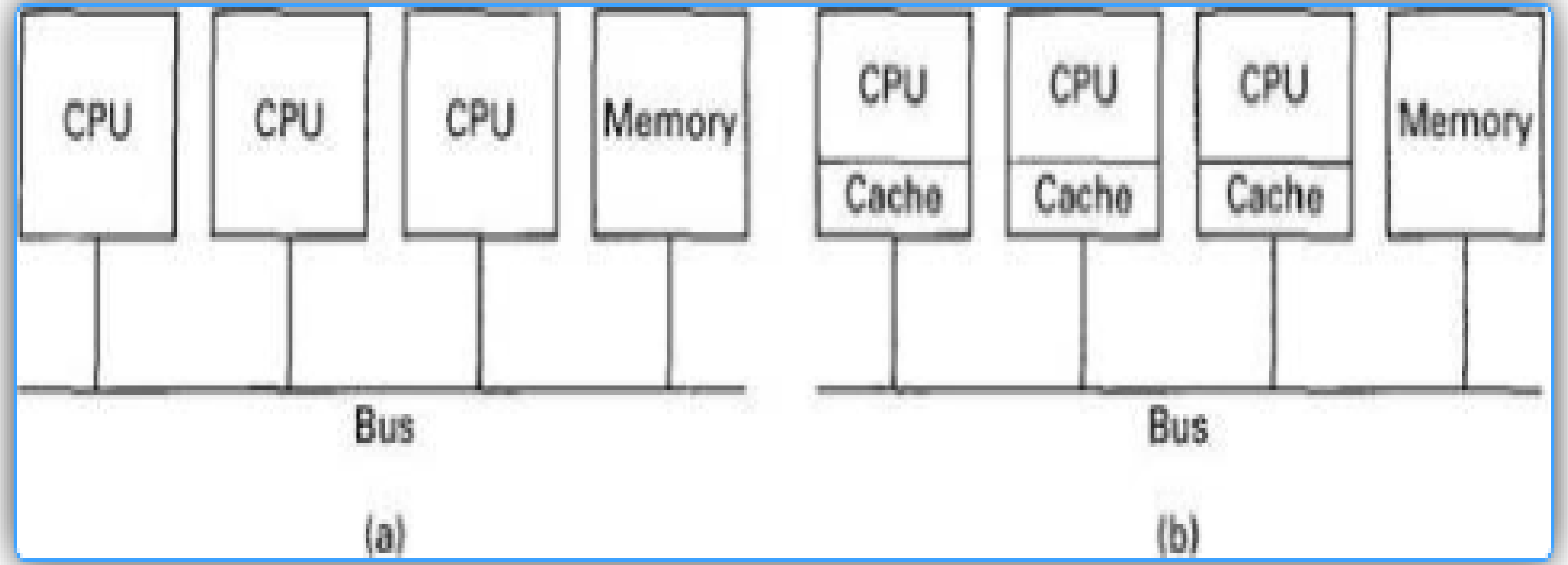


Fig. 6-2. (a) A multiprocessor. (b) A multiprocessor with caching.

Bus Contd..

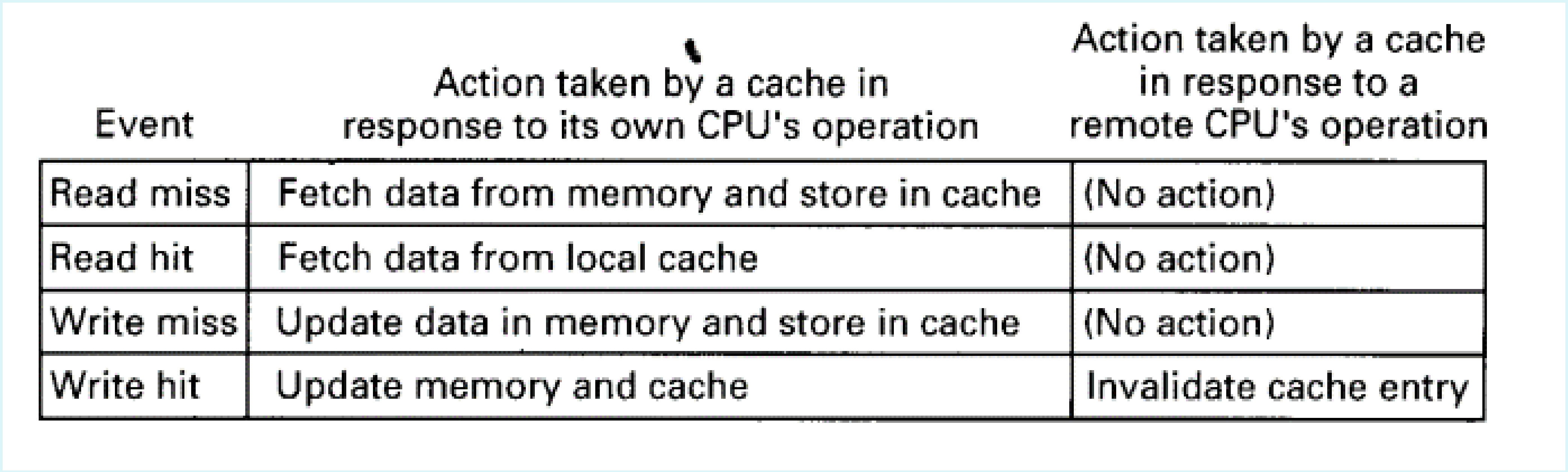
- A system with three CPUs and a memory shared among all of them.
- When any of the CPUs wants to read a word from the memory, it puts the address of the word it wants on the bus and asserts (puts a signal on) a bus control line indicating that it wants to do a read.
- When the memory has fetched the requested word, it puts the word on the bus and asserts another control line to announce that it is ready.
- The CPU then reads in the word. Writes work in an analogous way.

Bus Arbitration

- **To prevent two or more CPUs from trying to access the memory at the same time**, some kind of bus arbitration is needed.
- Various schemes are in use. For example, to acquire the bus, a CPU might first **have to request it by asserting a special request line**.
- **Only after receiving permission would it be allowed to use the bus**.
- The granting of this permission can be done in a centralized way, using a bus arbitration device, or in a decentralized way, with the first requesting CPU along the bus winning any conflict.

Write Through Cache Consistency protocol:

- One particularly simple and common protocol is called **write through.**
- When a **CPU first reads a word from memory**, that **word is fetched over the bus** and is **stored in the cache** of the CPU making the request.
- If that word is needed again later, the **CPU can take it from the cache without making a memory request**, thus reducing bus traffic.
- There are two cases, **read miss** (word not cached) and **read hit** (word cached) as the first two lines in the table. In simple systems, only the word requested is cached, but in most, a block of words of say, 16 or 32 words, is transferred and cached on the initial access and kept for possible future use.

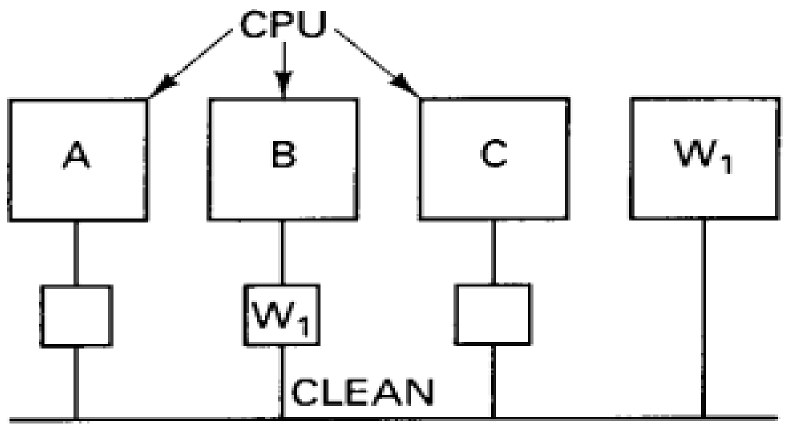


Event	Action taken by a cache in response to its own CPU's operation	Action taken by a cache in response to a remote CPU's operation
Read miss	Fetch data from memory and store in cache	(No action)
Read hit	Fetch data from local cache	(No action)
Write miss	Update data in memory and store in cache	(No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

States of a cache block

Our protocol manages cache blocks, each of which can be in one of the following three states:

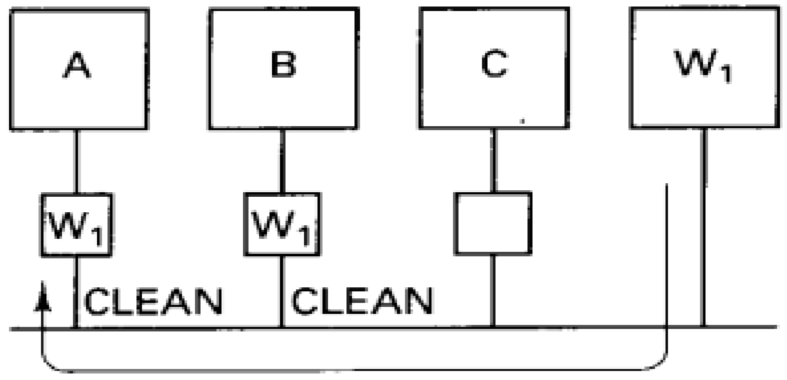
- 1. **INVALID** – This cache block does not contain valid data.
- 2. **CLEAN** – Memory is up-to-date; the block may be in other caches.
- 3. **DIRTY** – Memory is incorrect; no other cache holds the block.
- The basic idea is that a word that is being read by multiple CPUs is allowed to be present in all their caches.
- A word that is being heavily written by only one machine is kept in its cache and not written back to memory on every write to reduce bus traffic.



Memory is correct

Initial state.
Word W containing value W_1 is in memory and is also cached by B.

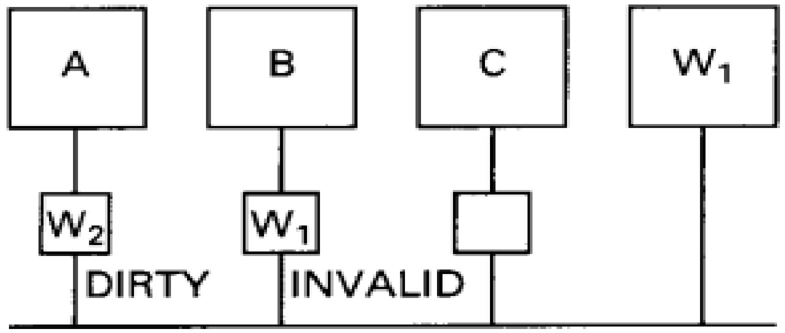
(a)



Memory is correct

A reads word W and gets W_1. B does not respond to the read, but the memory does.

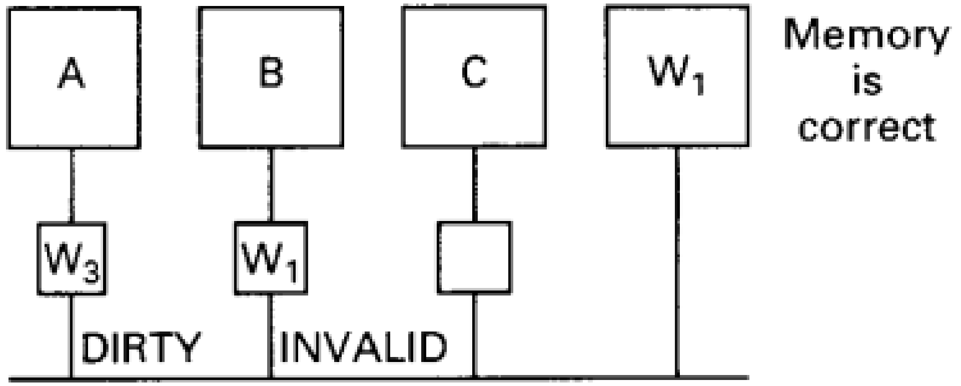
(b)



Memory is correct

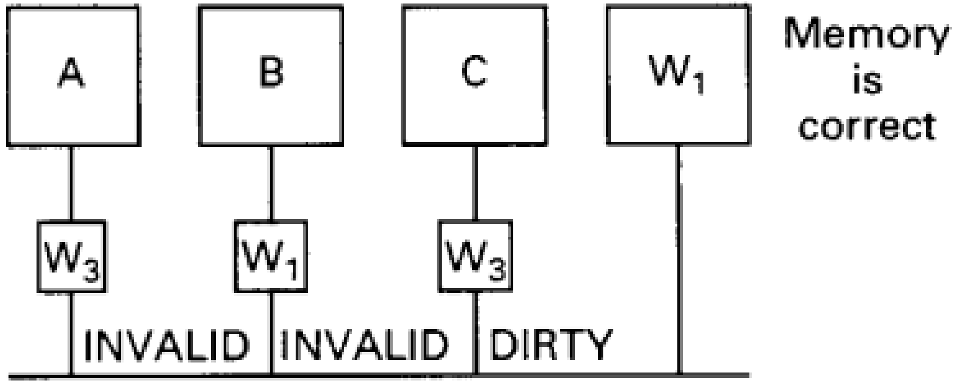
A writes a value W_2. B snoops on the bus, sees the write, and invalidates its entry. A's copy is marked DIRTY.

(c)



A writes W again. This and subsequent writes by A are done locally, without any bus traffic.

(d)



C reads or writes W. A sees the request by snooping on the bus, provides the value, and invalidates its own entry. C now has the only valid copy.

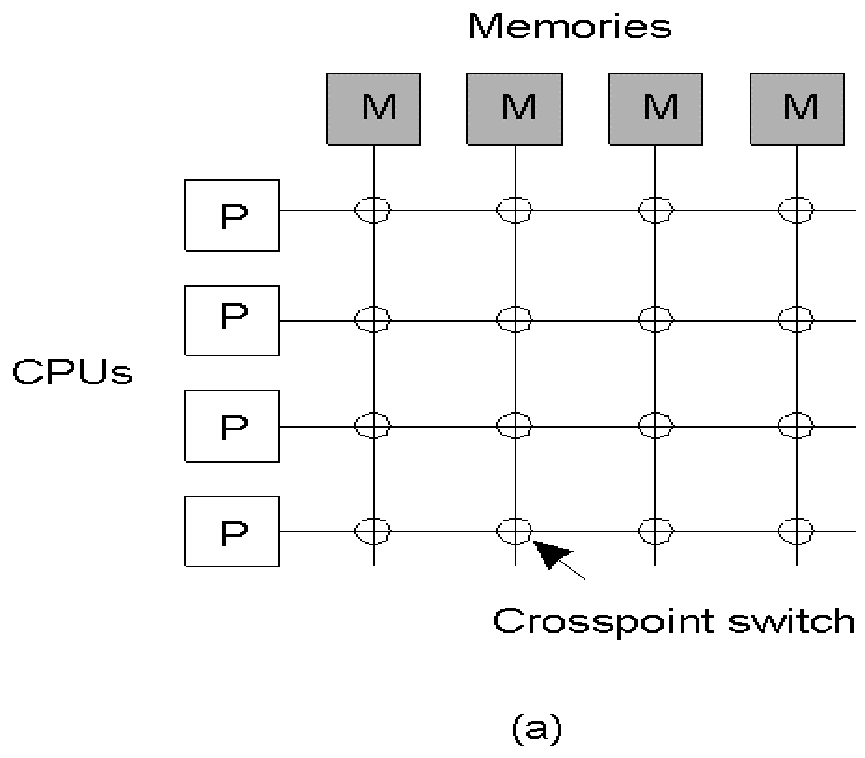
(e)

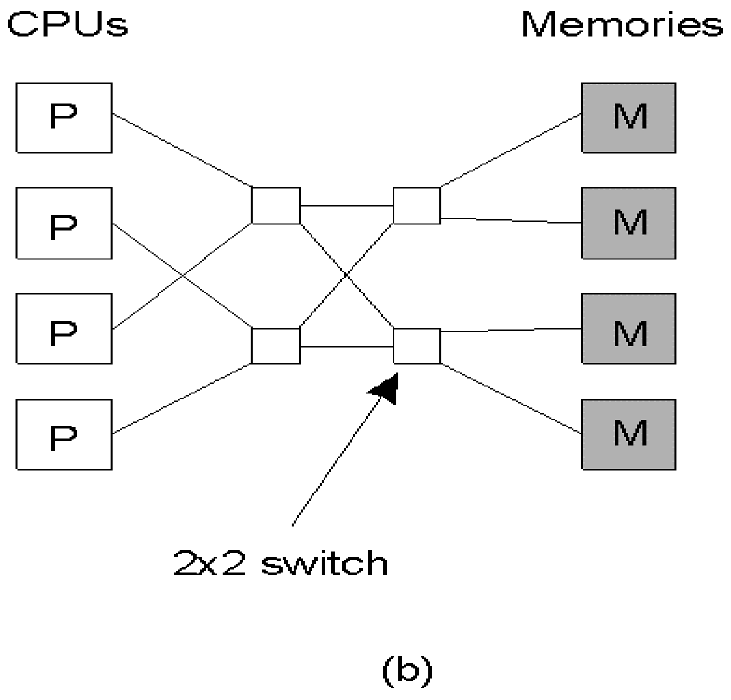
Fig. 6-4. An example of how a cache ownership protocol works.

Switched Multiprocessors

- To build a multiprocessor with more than 64 processors, a different method is needed to connect the CPUs with the memory.
- One possibility is to divide the memory up into modules and connect them to the CPUs with a **crossbar switch**
- Each CPU and each memory has a connection coming out of it.
- At every intersection is a tiny electronic **crosspoint switch** that can be opened and closed in hardware.
- When a CPU wants to access a particular memory, the crosspoint switch connecting them is closed momentarily, to allow the access to take place.
- The virtue of the crossbar switch is that many CPUs can be accessing memory at the same time, although if two CPUs try to access the same memory simultaneously, one of them will have to wait.

Cross bar switch and Omega switching network





A crossbar switch

An Omega Switching network

Crossbar Switch

- Useful to connect multiple CPUs with many memory modules
- Crosspoint switch is used to connect CPU with memory module based on requirement.
- To connect n-CPUs and n-Memory modules, n^2 crosspoint switches are required.
- For larger n, the interconnection is complex.

An omega switching network

- Contains 4 (2x2) switches having 2 input and 2 output ports, switching between them.
- Less complex compared to crossbar switch as this omega switching requires only n/2 switches to connect n-CPUs with n-memory modules with $\log_2 n$ switching stages.
- Limitations: for larger values of n, the switching time is larger and increases delay in instruction execution. Moreover its expensive

Omega Switching Networks

- This network contains four 2X2 switches, each having two inputs and two outputs.
- Each switch can route either input to either output.
- A careful look at the figure will show that with proper settings of the switches, every CPU can access every memory.
- These switches can be set in nanoseconds or less.
- In the general case, with *n* CPUs and *n* memories, the omega network requires $\log_2 n$ switching stages, each containing *n/2* switches, for a total of $(n \log_2 n)/2$ switches

Ring Based Multiprocessor

- The next step along the path toward distributed shared memory systems are ring-based multiprocessors, explained by **Memnet.**
- In Memnet, a single address space is divided into a **private part** and a **shared part**.
- The private part is divided up into **regions** so that **each machine has a piece for its stacks** and other **unshared data and code**.

Ring Based Multiprocessor

- The **shared part is common to all machines** (and distributed over them) and is **kept consistent** by a **hardware protocol** roughly similar to those used on bus-based multiprocessors.
- Shared memory is divided into 32-byte blocks, which is the unit in which transfers between machines take place.
- All the machines in Memnet are **connected together** in a modified **token-passing ring**.
- The ring consists of **20 parallel wires**, which together allow **16 data bits** and **4 control bits** to be sent **every 100 nsec**, for a **data rate of 160 Mbps**.

Memnet- Ring Based Multiprocessor

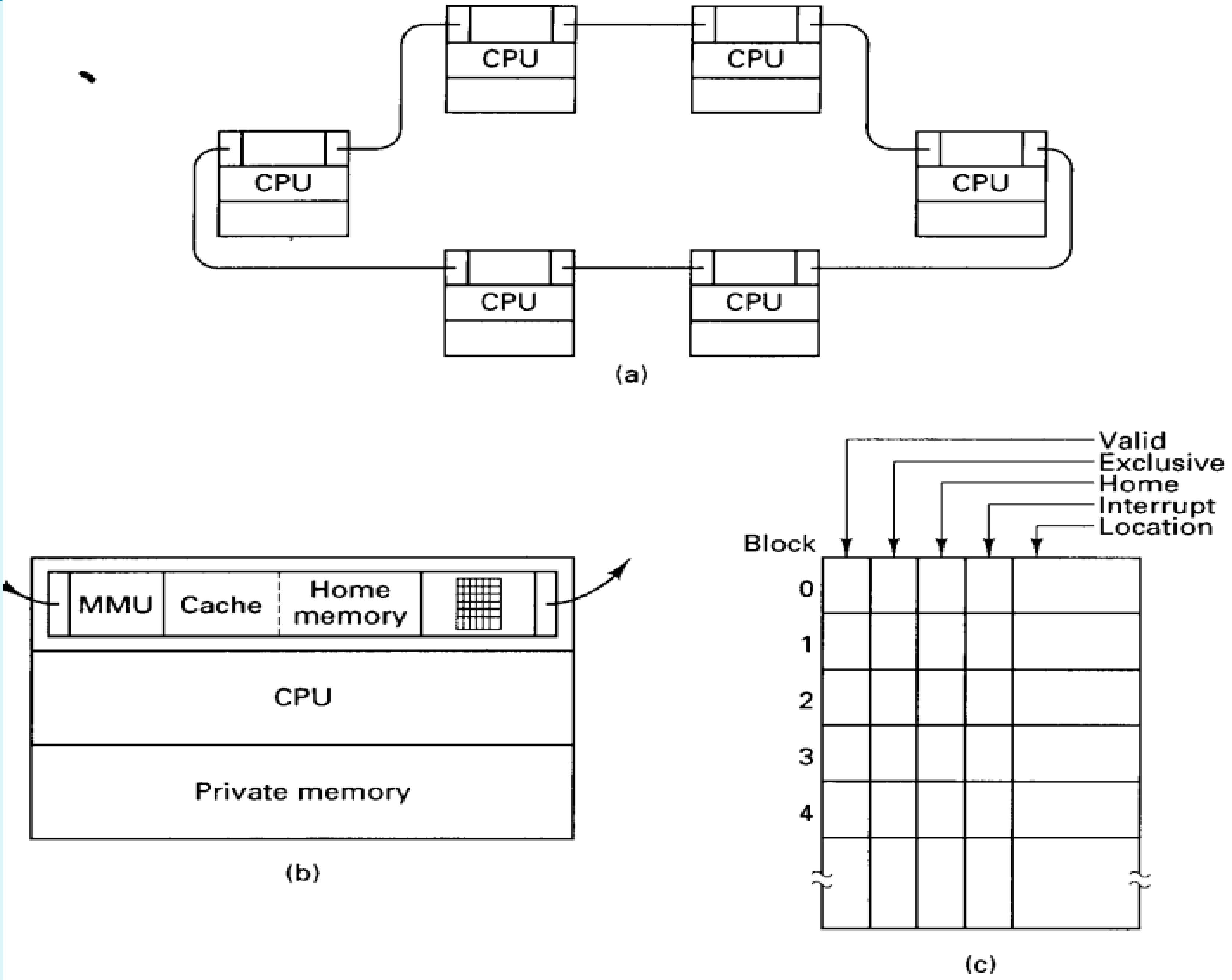


Fig. 6-5. (a) The Memnet ring. (b) A single machine. (c) The block table.

Memnet..Contd..

- In Memnet there is no centralized global memory.
- Instead, each 32-byte block in the shared address space has a home machine on which physical memory is always reserved for it, in the *Home memory* field .
- A block may be cached on a machine other than its home machine. (The cache and home memory areas share the same buffer pool, but since they are used slightly differently, we treat them here as separate entities.)
- A read-only block may be present on multiple machines; a read-write block may be present on only one machine.
- In both cases, a block need not be present on its home machine. All the home machine does is provide a guaranteed place to store the block if no other machine wants to cache it.
- This feature is needed because there is no global memory.
- In effect, the global memory has been spread out over all the machines.

Memnet-Block Table

- The Memnet device on each machine contains a table.
- It contains an entry for each block in the shared address space, indexed by block number.
- Each entry contains a
  - ***Valid* bit** telling whether the block is present in the cache and up to date,
  - ***Exclusive* bit**, specifying 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, and
  - ***Location* field** that tells where the block is located in the cache if it is present and valid.

NUMA (NonUniform Memory Access)

- People have attempted to reduce the cost by going to **hierarchical systems**.
- **Some memory is associated with each CPU**.
- Each **CPU can access its own local memory quickly**, but accessing anybody else's memory is slower.
- Non-uniform Memory Access is applicable for **real-time applications and time-critical applications**.
- In Non-Uniform Memory Access, **memory access time is not equal**.
- In non-uniform Memory Access, There are 2 types of buses used which are: Tree and hierarchical.

NUMA

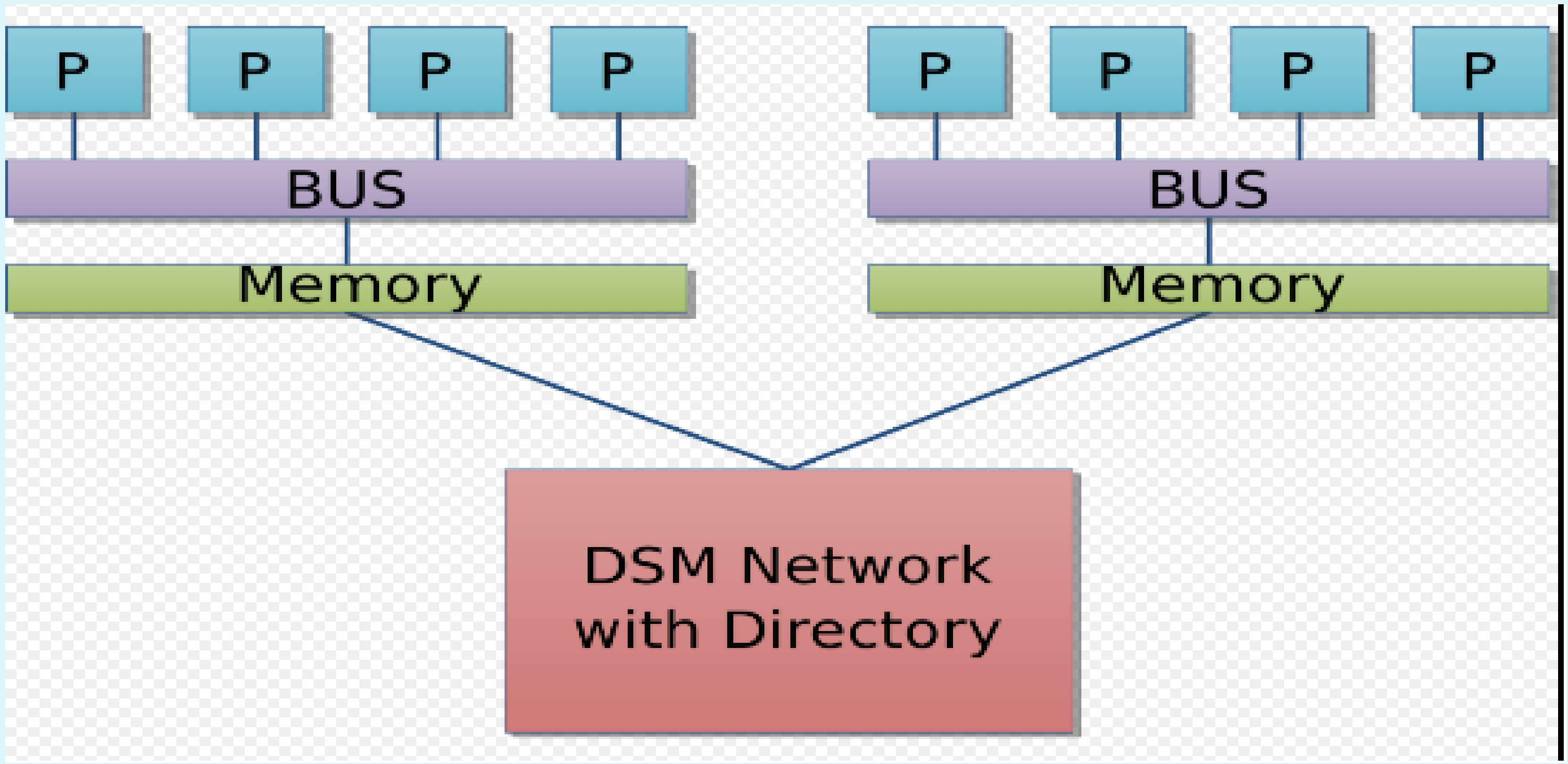
- A NUMA machine has a single virtual address space that is visible to all CPUs.
- When any CPU writes a value to location *a,* a subsequent read of a by a different processor will return the value just written.

Properties of NUMA Multiprocessors

NUMA machines have three key properties that are of concern to us:

1. Access to remote memory is possible.
2. Accessing remote memory is slower than accessing local memory.
3. Remote access times are not hidden by caching.

NUMA architecture



Comparison of Shared memory

- Shared memory systems cover a broad spectrum, from systems that maintain consistency entirely in hardware to those that do it entirely in software.
- we have the single-bus multiprocessors that have hardware caches and keep them consistent by snooping on the bus. These are the simplest shared-memory machines and operate entirely in hardware.
- Various machines made by Sequent and other vendors and the experimental DEC Firefly workstation (five VAXes on a common bus) fall into this category. This design works fine for a small or medium number of CPUs, but degrades rapidly when the bus saturates.
- Next come the switched multiprocessors, such as the Stanford Dash machine and the M.I.T. Alewife machine.
- These also have hardware caching but use directories and other data structures to keep track of which CPUs or clusters have which cache blocks.
- Complex algorithms are used to maintain consistency, but since they are stored primarily in MMU microcode (with exceptions potentially handled in software), they count as mostly "hardware" implementations.

Comparison of Shared memory

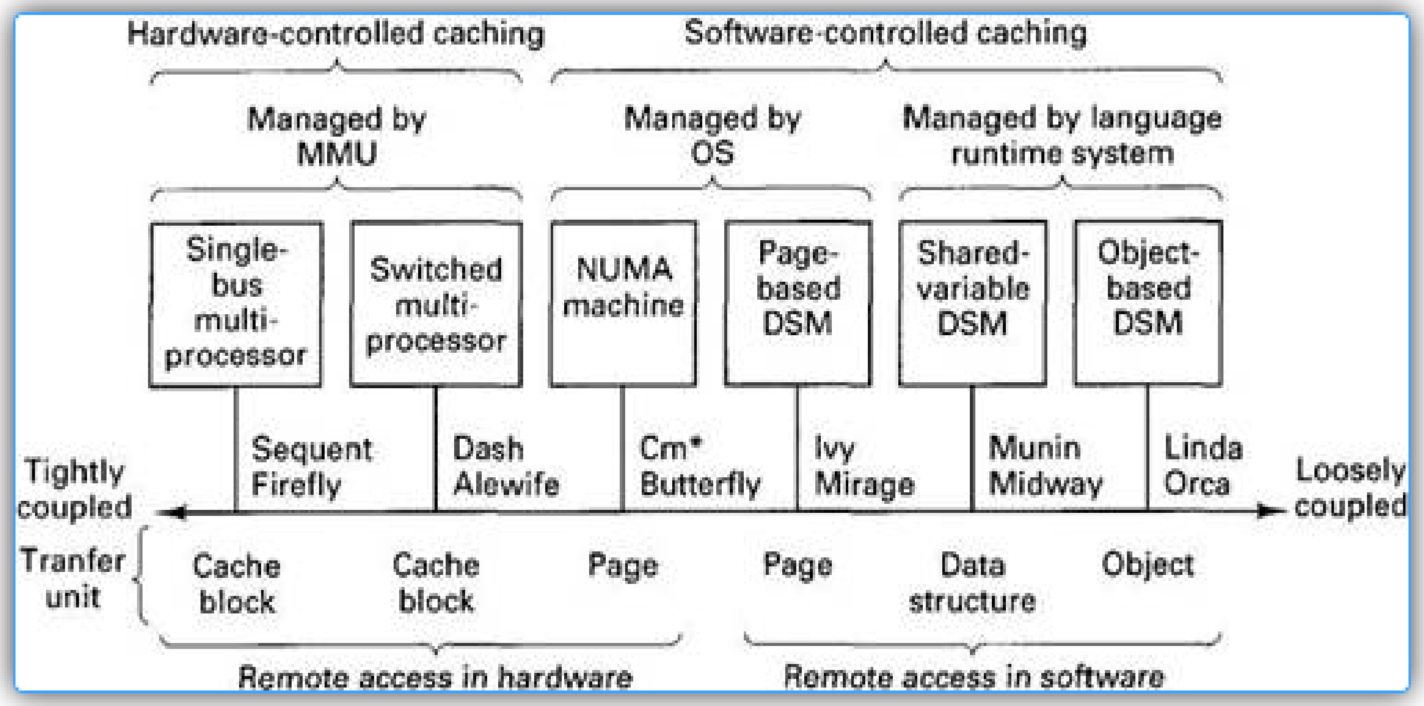
- Next come the NUMA machines. These are hybrids between hardware and software control. As in a multiprocessor, each NUMA CPU can access each word of the common virtual address space just by reading or writing it.
- Unlike in a multiprocessor, however, caching (i.e., page placement and migration) is controlled by software (the operating system), not by hardware (the MMUs).
- BBN Butterfly are examples of NUMA machines.

- Continuing along the spectrum, we come to the machines running a page-based distributed shared memory system such as IVY and Mirage
- Each of the CPUs in such a system has its own private memory and, unlike the NUMA machines and UMA multiprocessors, cannot reference remote memory directly.
- When a CPU addresses a word in the address space that is backed by a page currently located on a different machine, a trap to the operating system occurs and the required page must be fetched by software.
- The operating system acquires the necessary page by sending a message to the machine where the page is currently residing and asking for it. Thus both placement and access are done in software here.

Comparison of Shared memory

- Finally, we have systems running object-based distributed shared memory. Unlike all the others, programs here cannot just access the shared data.
- They have to go through protected methods, which means that the runtime system can always get control on every access to help maintain consistency.
- Everything is done in software here, with no hardware support at all.
- Orca is an example of this design, and Linda is similar to it in some important ways.

Broad view of shared machines



. The spectrum of shared memory machines.

Consistency Models

Strict consistency - Casual consistency

PRAM consistency - Weak consistency

Release consistency - Entry consistency

Consistency Model

- A **consistency model** is essentially a contract between the software and the memory.
- It says that if the software agrees to obey certain rules, the memory promises to work correctly.
- If the software violates these rules, all bets are off and correctness of memory operation is no longer guaranteed.
- A wide spectrum of contracts have been devised, ranging from contracts that place only minor restrictions on the software to those that make normal programming nearly impossible.

Strict Consistency

- The most stringent consistency model is called **strict consistency**. It is defined by the following condition:
- *Any read to a memory location x returns the value stored by the most recent write operation to x.*
- When memory is strictly consistent, **all writes are instantaneously visible to all processes** and an **absolute global time order** is maintained.
- If a **memory location is changed, all subsequent reads from that location see the new value**, no matter how soon after the change the reads are done and no matter which processes are doing the reading and where they are located.
- Similarly, **if a read is done, it gets the then-current value**, no matter how quickly the next write is done.

Example

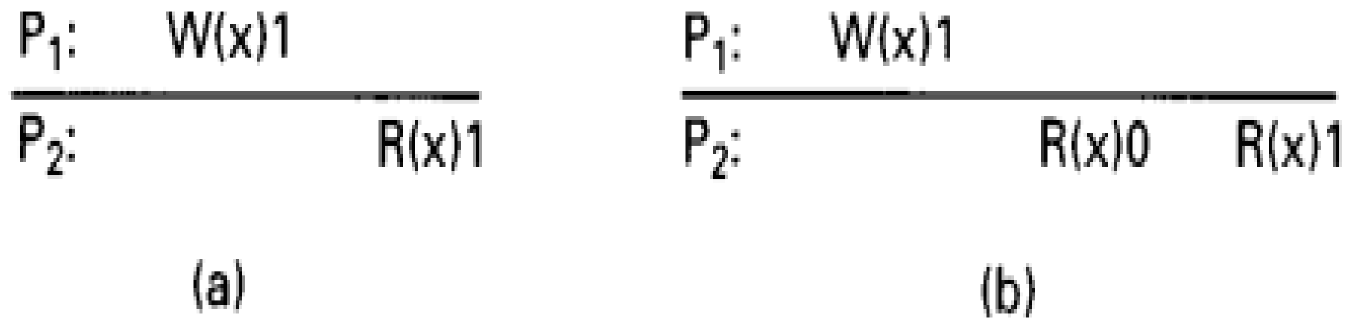


Fig. 6-12. Behavior of two processes. The horizontal axis is time. (a) Strictly consistent memory. (b) Memory that is not strictly consistent.

Example

- Several processes, ***P1* , P*2*, and so on** can be shown at different heights in the figure.
- The operations done by each process are shown horizontally, with time increasing to the right.
- Straight lines separate the processes.
- The symbols *W(x)a* and *R(y)b*

  mean that **write to *x* with the value *a*** and **read from *y* returning *b*** have been done, respectively.
- The initial value of all variables in these diagrams is assumed to be 0.
- As an example, P_1 does a write to location *x,* storing the value 1.
- Later, P_2 reads *x* and sees the 1.
- This behaviour is correct for a strictly consistent memory.

Example

- Suppose *x* is a variable stored only on machine *B.*
- Imagine that a process on machine *A* reads *x* at time T_1, which means that a message is then sent to *B* to get *x.*
- Slightly later, at T_2, a process on *B* does a write to *x.* If strict consistency holds, the read should always return the old value regardless of where the machines are and how close T_2 is to T_1

Causal Consistency

- For a memory to be considered causally consistent, it is necessary that the memory obey the following condition:
- *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.*
- The **causal consistency** model represents a consistency that it makes a distinction between events that are potentially causally related and those that are not.
- If **event *B* is caused or influenced by** an earlier **event, *A*,** causality requires that everyone else first **see *A*, then see *B*.**

Example:

- Suppose that process P_1 **writes a variable** x.
- Then P_2 **reads** x **and writes** y. Here the **reading of** x **and the writing of** y **are potentially causally related** because the computation of y may have depended on the value of x read by P_2(i.e., the value written by P_1).
- On the other hand, if **two processes spontaneously and simultaneously write two variables, these are not causally related**.
- When there is a **read followed later by a write**, the two events are potentially causally related.
- Similarly, a read is causally related to the write that provided the data the read got. Operations that are not causally related are said to be **concurrent.**

PRAM Consistency

- In causal consistency, it is permitted that concurrent writes be seen in a different order on different machines, although causally-related ones must be seen in the same order by all machines.
- The next step in relaxing memory is to drop the latter requirement.
- Doing so gives **PRAM consistency** (Pipelined RAM), which is subject to the condition:
- *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.*

Weak Consistency

- Consider the case of a process inside a critical section reading and writing some variables in a tight loop.
- Even though other processes are not supposed to touch the variables until the first process has left its critical section, the memory has no way of knowing when a process is in a critical section and when it is not, so it has to propagate all writes to all memories in the usual way.
- A better solution would be to let the process finish its critical section and then make sure that the final results were sent everywhere, not worrying too much whether all intermediate results had also been propagated to all memories in order, or even at all.
- This can be done by introducing a new kind of variable, a **synchronization variable**, that is used **for synchronization purposes**, the operations on it are used to **synchronize memory**.
- When a synchronization completes, all writes done on that machine are propagated outward and all writes done on other machines are brought in.
- In other words, all of (shared) memory is synchronized.

Properties of Weak Consistency

We define this model, called **weak consistency**, by saying

- *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.*

Weak Consistency Models

- Point 1 says that all processes see all accesses to synchronization variables in the same order. Effectively, when a synchronization variable is accessed, this fact is broadcast to the world, and no other synchronization variable can be accessed in any other process until this one is finished everywhere.
- Point 2 says that accessing a synchronization variable "flushes the pipeline." It forces all writes that are in progress or partially completed or completed at some memories but not others to complete everywhere. When the synchronization access is done, all previous writes are guaranteed to be done as well. By doing a synchronization after updating shared data, a process can force the new values out to all other memories.
- Point 3 says that when ordinary (i.e., not synchronization) variables are accessed, either for reading or writing, all previous synchronizations have been performed. By doing a synchronization before reading shared data, a process can be sure of getting the most recent values.

Release Consistency

- Weak consistency has the problem that when a synchronization variable is accessed, the memory does not know whether this is being done because the process is finished writing the shared variables or about to start reading them.
- Consequently, it must take the actions required in both cases, namely making sure that all locally initiated writes have been completed (i.e., propagated to all other machines), as well as gathering in all writes from other machines.
- If the memory could tell the difference between entering a critical region and leaving one, a more efficient implementation might be possible. To provide this information, two kinds of synchronization variables or operations are needed instead of one.

Acquire and Release

Release consistency provides these two kinds.

- **Acquire** accesses are used to tell the memory system that a critical region is about to be entered.
- **Release** accesses say that a critical region has just been exited. These accesses can be implemented either as ordinary operations on special variables or as special operations.
- In either case, the programmer is responsible for putting explicit code in the program telling when to do them, for example, by calling library procedures such as *acquire* and *release* or procedures such as *enter_critical_region* and *leave_critical_region.*

Rules for Release consistency

- In general, a distributed shared memory is release consistent if it 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).*
- If all these conditions are met and processes use acquire and release properly (i.e., in acquire-release pairs), the results of any execution will be no different than they would have been on a sequentially consistent memory. In effect, blocks of accesses to shared variables are made atomic by the acquire and release primitives to prevent interleaving.

Entry Consistency

- Another consistency model that has been designed to be used with critical sections is entry consistency.
- Like both variants of release consistency, it requires the programmer (or compiler) to use acquire and release at the start and end of each critical section, respectively.
- However, unlike release consistency, entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier.
- If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks.

Entry consistency

- When an acquire is done on a synchronization variable, only those ordinary shared variables guarded by that synchronization variable are made consistent.
- Entry consistency differs from lazy release consistency in that the latter does not associate shared variables with locks or barriers and at acquire time has to determine empirically which variables it needs.
- Associating with each synchronization variable a list of shared variables reduces the overhead associated with acquiring and releasing a synchronization variable, since only a few shared variables have to be synchronized.
- It also allows multiple critical sections involving disjoint shared variables to execute simultaneously, increasing the amount of parallelism. The price paid is the extra overhead and complexity of associating every shared data variable with some synchronization variable.

Contd..

- Programming this way is also more complicated and error prone. Synchronization variables are used as follows.
- Each synchronization variable has a current owner, namely, the process that last acquired it. The owner may enter and exit critical regions repeatedly without having to send any messages on the network.
- A process not currently owning a synchronization variable but wanting to acquire it has to send a message to the current owner asking for ownership and the current values of the associated variables.
- It is also possible for several processes simultaneously to own a synchronization variable in nonexclusive mode, meaning that they can read, but not write, the associated data variables

Contd..

- Formally, a memory exhibits entry consistency if it 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.

Contd..

- The first condition says that when a process does an acquire, the acquire may not complete (i.e., return control to the next statement) until all the guarded shared variables have been brought up to date. In other words, at an acquire, all remote changes to the guarded data must be made visible.
- The second condition says that before updating a shared variable, a process must enter a critical region in exclusive mode to make sure that no other process is trying to update it at the same time.
- The third condition says that if a process wants to enter a critical region in nonexclusive mode, it must first check with the owner of the synchronization variable guarding the critical region to fetch the most recent copies of the guarded shared variables.

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.

Page Based Distributed Shared Memory

- Workstations on a LAN are fundamentally different from a multiprocessor.
- Processors can only reference their own local memory.
- There is no concept of a global shared memory, as there is with a NUMA or UMA multiprocessor.
- The goal of the DSM work, however, is to add software to the system to allow a multicomputer to run multiprocessor programs. Consequently, when a processor references a remote page, that page must be fetched.

Page Based Distributed Shared Memory- Basic Design

- The idea behind DSM is simple: try to **emulate the cache** of a multiprocessor **using the MMU and operating system software**.
- In a DSM system, the **address space is divided up into chunks**, with the **chunks being spread over all the processors in the system**.

Page Based Distributed Shared Memory Basic Design

- 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.

Replication

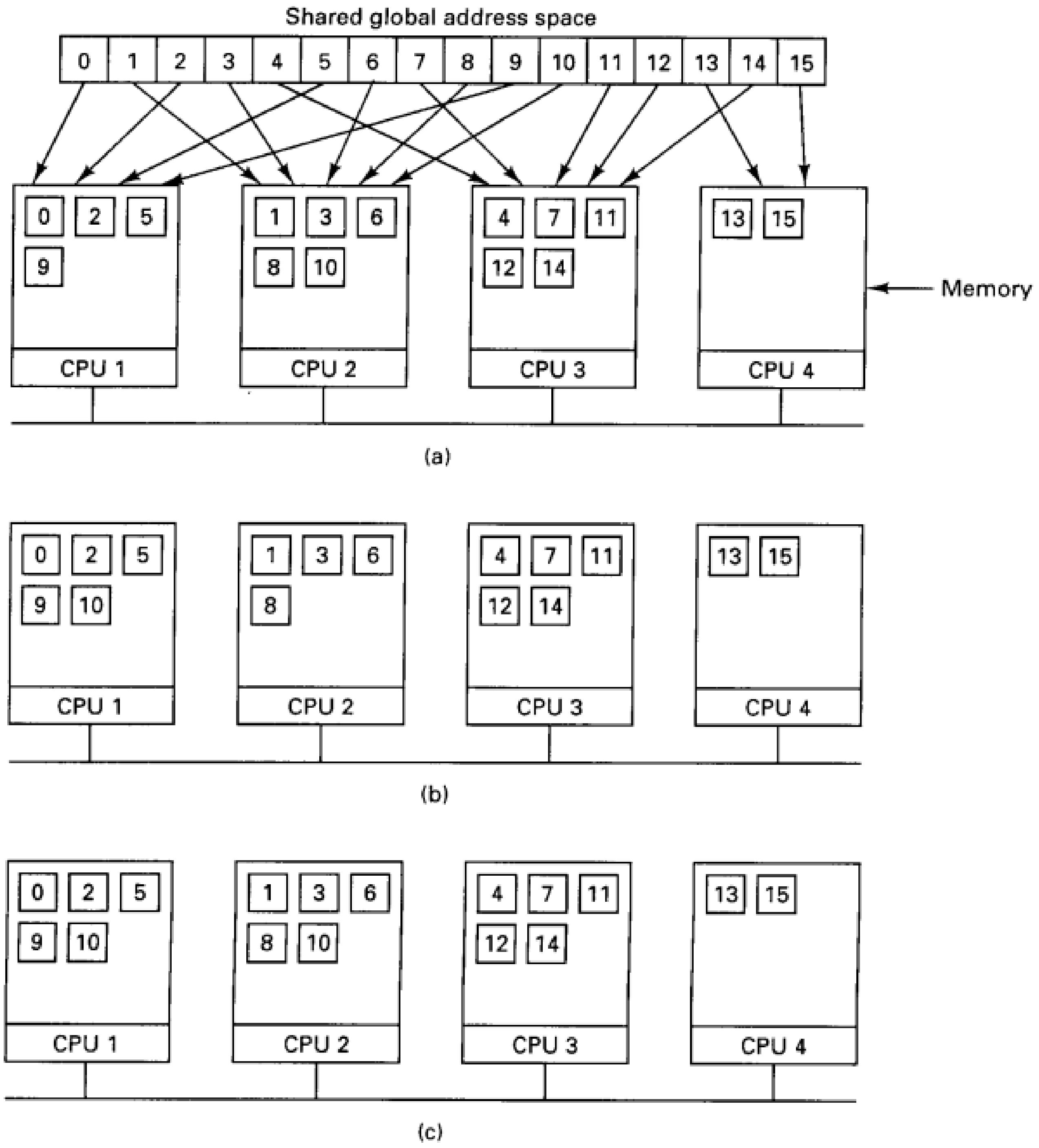


Fig. 6-25. (a) Chunks of address space distributed among four machines. (b) Situation after CPU 1 references chunk 10. (c) Situation if chunk 10 is read only and replication is used.

Contd…

- This concept is illustrated in Fig. 1(a) for an address space with 16 chunks and four processors.
- 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. 1(b).

Granularity

- When a process references a word that is absent, it causes a page fault.
- An obvious choice is to bring in the entire page that is needed.
- Furthermore, integrating DSM with virtual memory makes the total design simpler, since the same unit, the page, is used for both.
- On a page fault, the missing page is just brought in from another machine instead of from the disk, so much of the page fault handling code is the same as in the traditional case.

Granularity

- However, another possible choice is to bring in a larger unit, say a region of 2, 4, or 8 pages, including the needed page. In effect, doing this simulates a larger page size. There are advantages and disadvantages to a larger chunk size for DSM.

Granularity..contd..

- The biggest advantage is that because the startup time for a network transfer is substantial, it does not take much longer to transfer 1024 bytes than it does to transfer 512 bytes. By transferring data in large units, **when a large piece of address space has to be moved, the number of transfers may often be reduced.**
- This property is especially important because many programs exhibit locality of reference, meaning that if a program has referenced one word on a page, it is likely to reference other words on the same page in the immediate future
- On the other hand, the network will be tied up longer with a larger transfer, blocking other faults caused by other processes.
- Also, **too large an effective page size introduces a new problem, called false sharing,** illustrated in Fig. 6-26.
- Here we have a page containing two unrelated shared variables, A and B. Processor 1 makes heavy use of A, reading and writing it.
- Similarly, process 2 uses B. Under these circumstances, the page containing both variables will constantly be travelling back and forth between the two machines.

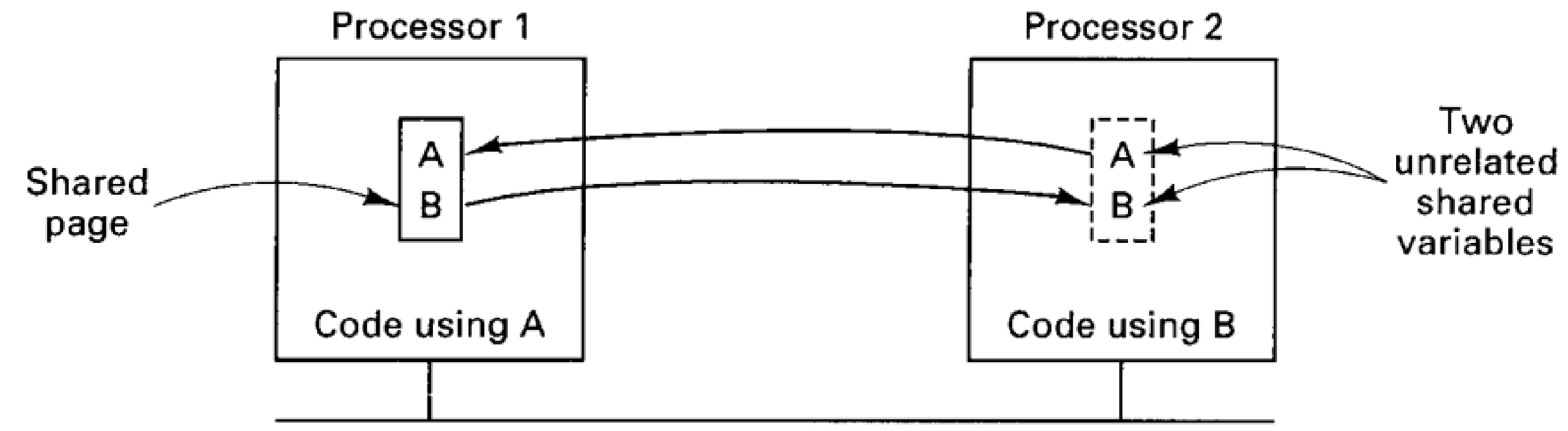
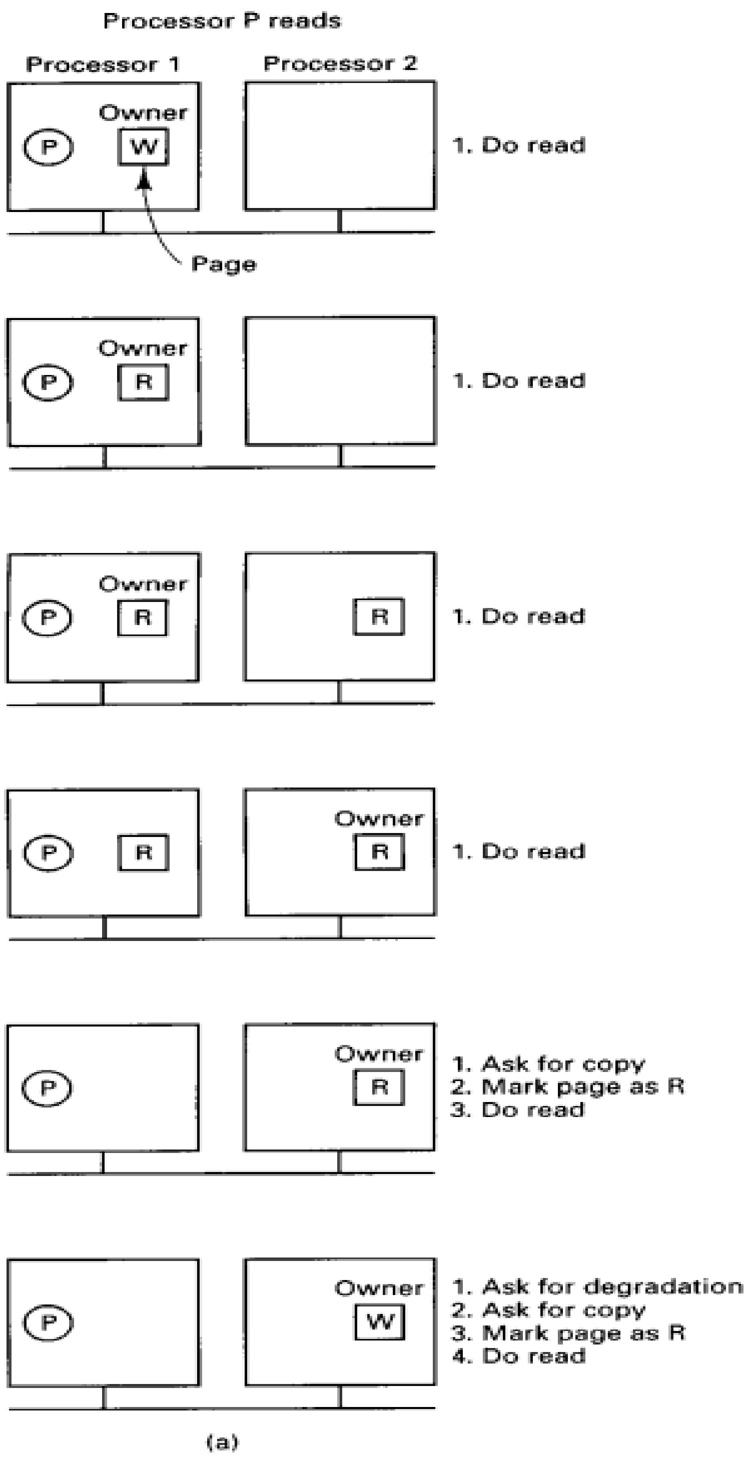


Fig. 6-26. False sharing of a page containing two unrelated variables.

The problem here is that although the variables are unrelated, since they appear by accident on the same page, when a process uses one of them, it also gets the other. The larger the effective page size, the more often false sharing will occur, and conversely, the smaller the effective page size, the less often it will occur. Nothing analogous to this phenomenon is present in ordinary virtual memory systems. Clever compilers that understand the problem and place variables in the address space accordingly can help reduce false sharing and improve performance. However, saying this is easier than doing it. Furthermore, if the false sharing consists of processor 1 using one element of an array and processor 2 using a different element of the same array, there is little that even a clever compiler can do to eliminate the problem.



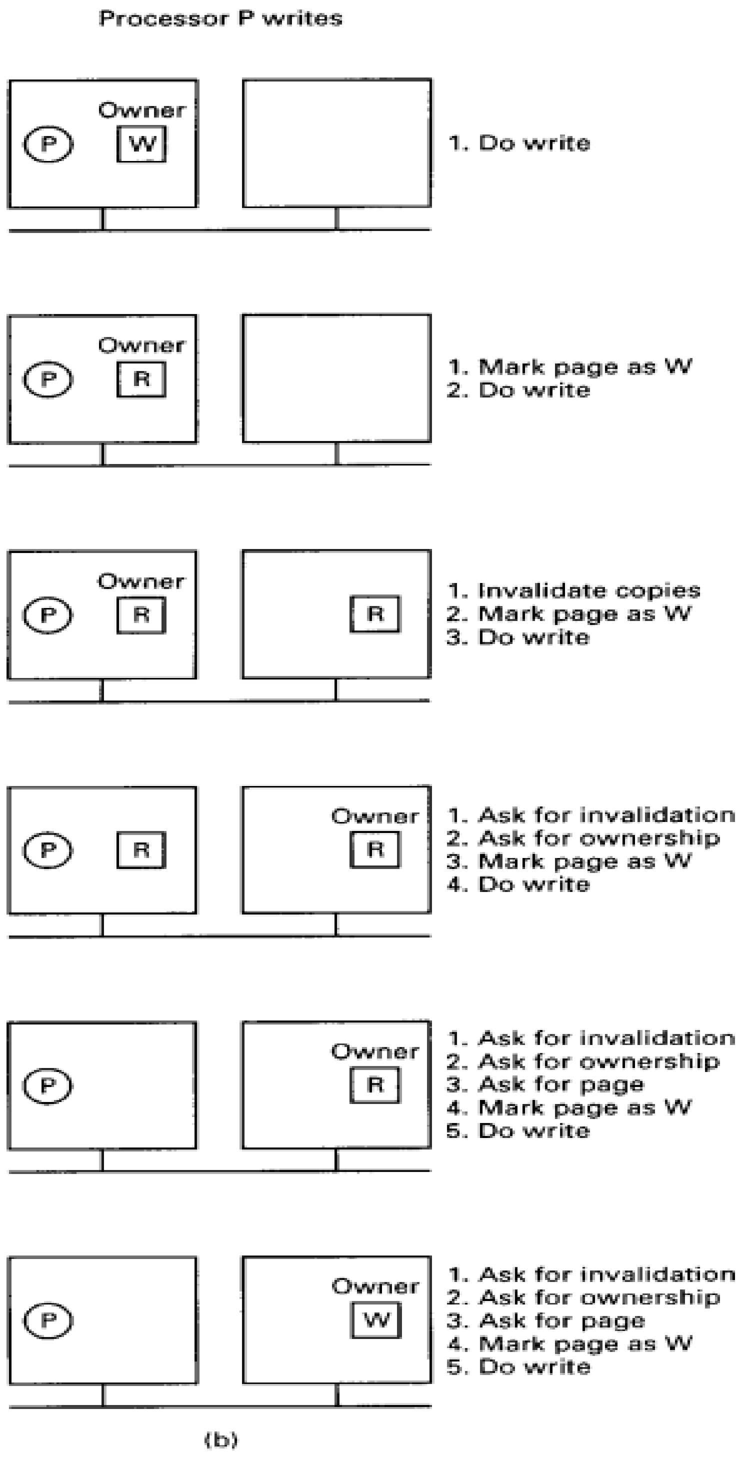


Fig. 6-27. (a) Process *P* wants to read a page. (b) Process *P* wants to write a page.

Finding the owner
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. **It is the job of the manager to keep track of who owns each page.**

Finding the owner
How to find the owner of the 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

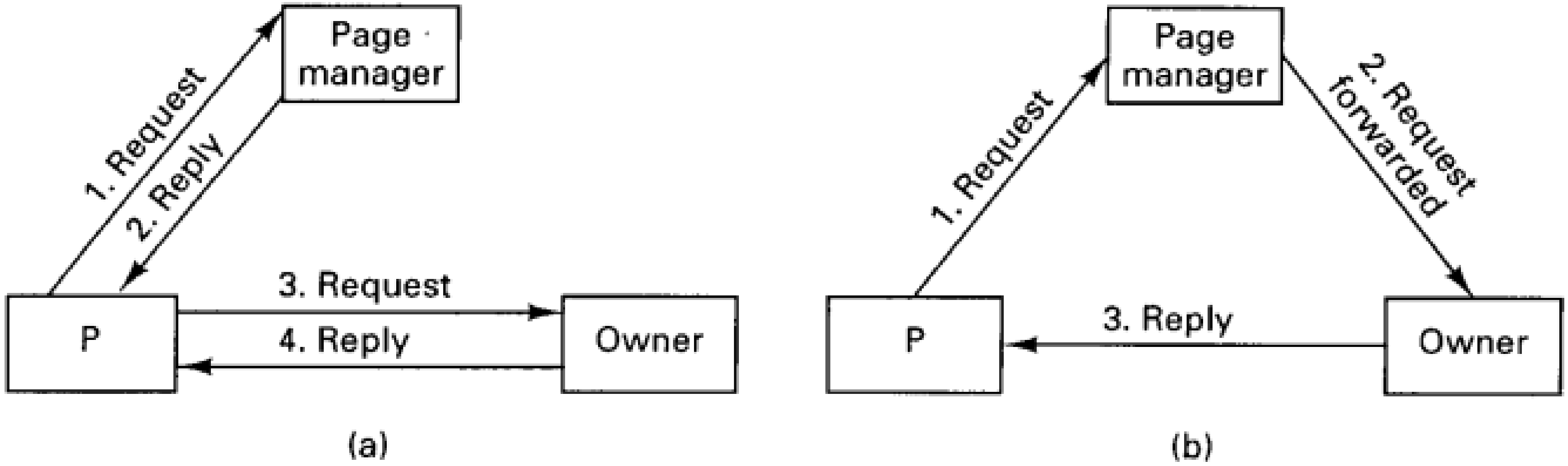


Fig. 6-28. Ownership location using a central manager. (a) Four-message protocol. (b) Three-message protocol.

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

Finding the Copies

- 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.

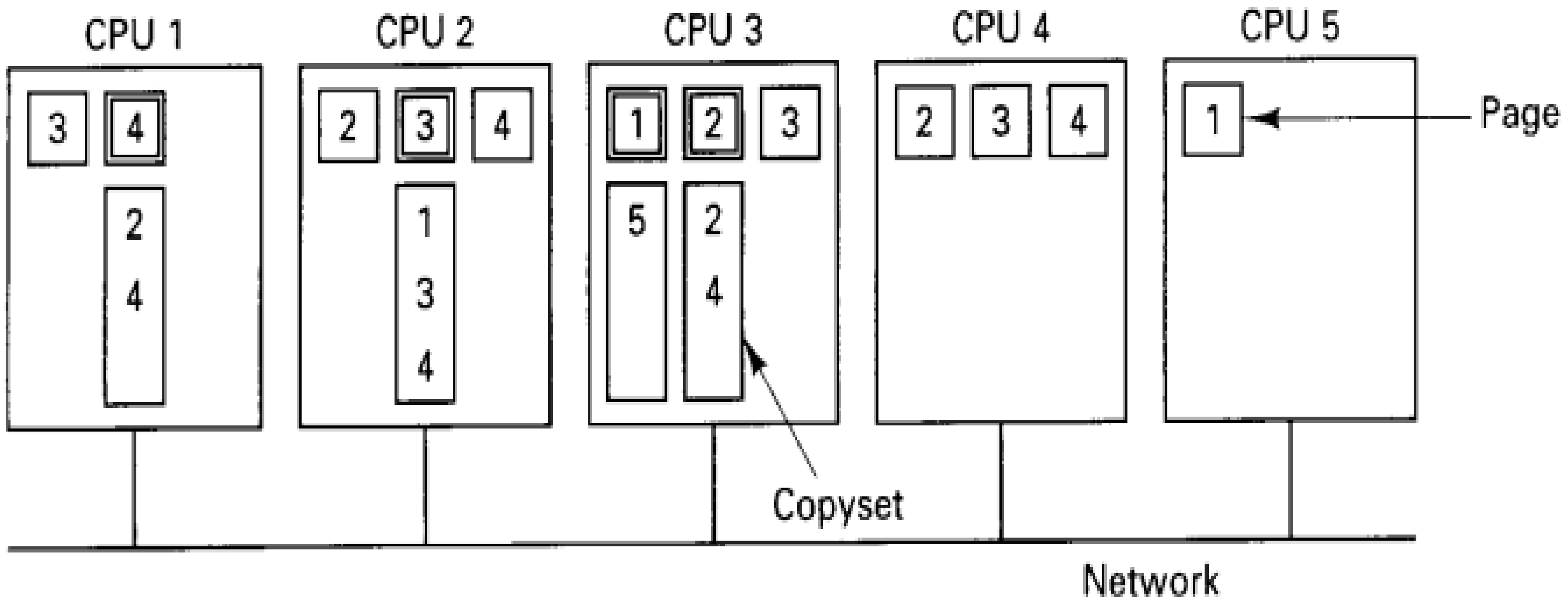


Fig. 6-29. The owner of each page maintains a copyset telling which other CPUs are sharing that page. Page ownership is indicated by the double boxes.

Contd..

- 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. Dash and Memnet also need to invalidate pages when a new writer suddenly appears, but they do it differently. Dash uses directories.
- The writing process sends a packet to the directory (the page manager in our terminology), which then finds all the copies from its bit map, sends each one an invalidation packet, and collects all the acknowledgements.
- Memnet fetches the needed page and invalidates all copies by broadcasting an invalidation packet on the ring. The first processor having a copy puts it in the packet and sets a header bit saying it is there.
- Subsequent processors just invalidate their copies. When the packet comes around the ring and arrives back at the sender, the needed data are present and all other copies are gone. In effect, Memnet implements DSM in hardware.

Page Replacement

- In a DSM system, as in any system using virtual memory, it can happen that a page is needed but that there is no free page frame in memory to hold it.
- When this situation occurs, a page must be evicted from memory to make room for the needed page. Two subproblems immediately arise: which page to evict and where to put it. To a large extent, the choice of which page to evict can be made using traditional virtual memory algorithms, such as some approximation to the least recently used (LRU) algorithm.

Page Replacement

- One complication that occurs with DSM is that pages can be invalidated spontaneously (due to the activities of other processes), which affects the possible choices. However, by maintaining the estimated LRU order of only those pages that are currently valid, any of the traditional algorithms can be used. As with conventional algorithms, it is worth keeping track of which pages are "clean" and which are "dirty."
- In the context of DSM, a replicated page that another process owns is always a prime candidate to evict because it is known that another copy exists. Consequently, the page does not have to be saved anywhere. If a directory scheme is being used to keep track of copies, the owner or page manager must be informed of this decision, however. If pages are located by broadcasting, the page can just be discarded. The second best choice is a replicated page that the evicting process owns. It is sufficient to pass ownership to one of the other copies by informing that process, the page manager, or both, depending on the implementation

Contd…

- The page itself need not be transferred, which results in a smaller message. If no replicated pages are suitable candidates, a nonreplicated page must be chosen, for example, the least recently used valid page.
- There are two possibilities for getting rid of it. The first is to write it to a disk, if present. The other is to hand it off to another processor. Choosing a processor to hand a page off to can be done in several ways.
- For example, each page could be assigned a home machine, which must accept it, although this probably implies reserving a large amount of normally wasted space to hold pages that might be sent home some day. Alternatively, the number of free page frames could be piggybacked on each message sent, with each processor building up an idea of how free memory was distributed around the network.

Contd…

- An occasional broadcast message giving the exact count of free page frames could help keep these numbers up to date. As an aside, note that a conflict may exist between choosing a replicated page (which may just be discarded) and choosing a page that has not been referenced in a long time (which may be the only copy). The same problem exists in traditional virtual memory systems, however, so the same compromises and heuristics apply.
- One problem that is unique to DSM systems is the network traffic generated when processes on different machines are actively sharing a writable page, either through false sharing or true sharing.

SHARED-VARIABLE DISTRIBUTED SHARED MEMORY

- Page-based DSM takes a normal linear address space and allows the pages to migrate dynamically over the network on demand. Processes can access all of memory using normal read and write instructions and are not aware of when page faults or network transfers occur.
- Accesses to remote data are detected and protected by the MMU. A more structured approach is to share only certain variables and data structures that are needed by more than one process.
- In this way, the problem changes from how to do paging over the network to how to maintain a potentially replicated, distributed data base consisting of the shared variables. Different techniques are applicable here, and these often lead to major performance improvements.
- The first question that must be addressed is whether or not shared variables will be replicated, and if so, whether fully or partially.
- If they are replicated, there is more potential for using an update algorithm rather than on a page-based DSM system, provided that writes to individual shared variables can be isolated.

OBJECT-BASED DISTRIBUTED SHARED MEMORY

- The page-based DSM systems that we studied use the MMU hardware to trap accesses to missing pages.
- While this approach has some advantages, it also has some disadvantages. In particular, in many programming languages, data are organized into objects, packages, modules, or other data structures, each of which has an existence independent of the others.
- If a process references part of an object, in many cases the entire object will be needed, so it makes sense to transport data over the network in units of objects, not in units of pages.

OBJECT-BASED DISTRIBUTED SHARED MEMORY

- The shared-variable approach, as taken by Munin and Midway, is a step in the direction of organizing the shared memory in a more structured way, but it is only a first step.
- In both systems, the programmer must supply information about which variables are shared and which are not, and must also provide protocol information in Munin and association information in Midway.
- Errors in these annotations can have serious consequences. By going further in the direction of a high-level programming model, DSM systems can be made easier and less error prone to program. Access to shared variables and synchronization using them can also be integrated more cleanly.
- In some cases, certain optimizations can also be introduced that are more difficult to perform in a less abstract programming model.

Objects

- An object is a programmer-defined encapsulated data structure, as depicted in Fig. 6-34.
- **It consists of internal data, the object state, and procedures, called methods or operations, that operate on the object state**.
- **To access or operate on the internal state, the program must invoke one of the methods**. The method can change the internal state, return (part of) the state, or something else.
- **Direct access to the internal state is not allowed. This property, called information hiding Forcing all references to an object's data to go through the methods helps structure the program in a modular way.**

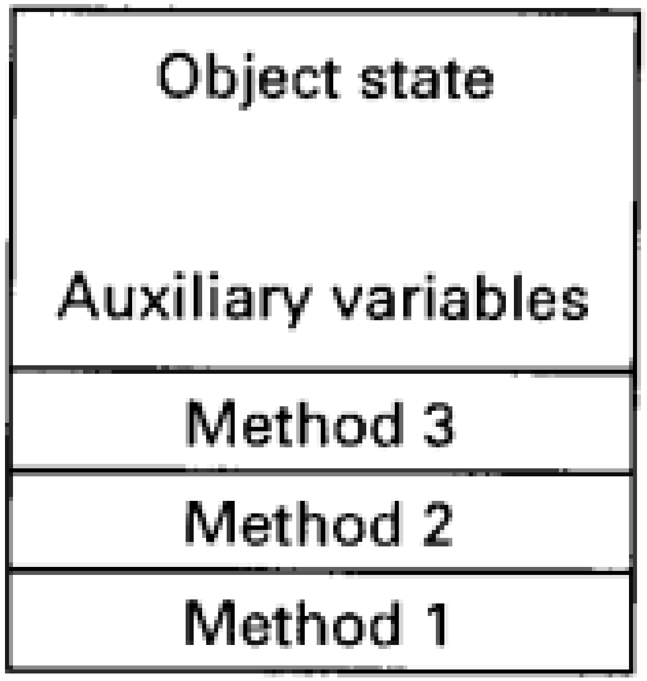


Fig. 6-34. An object.

Process Communication

- In an object-based distributed shared memory, processes on multiple machines share an abstract space filled with shared objects, as shown in Fig. 6-35.
- The location and management of the objects is handled automatically by the runtime system. This model is in contrast to page-based DSM systems such as IVY, which just provide a raw linear memory of bytes from 0 to some maximum.
- Any process can invoke any object's methods, regardless of where the process and object are located.
- It is the job of the operating system and runtime system to make the act of invoking a method work no matter where the process and object are located. Because processes cannot directly access the internal state of any of the shared objects, various optimizations are possible here that are not possible (or at least are more difficult) with page-based DSM.
- For example, since access to the internal state is strictly controlled, it may be possible to relax the memory consistency protocol without the programmer even knowing it.

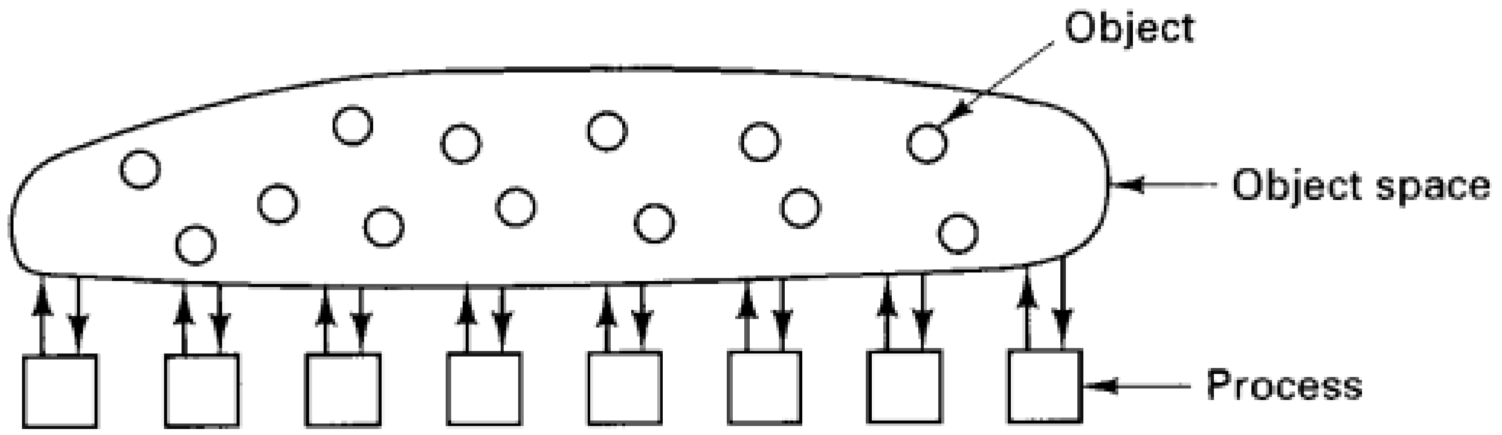


Fig. 6-35. In an object-based distributed shared memory, processes communicate by invoking methods on shared objects.

DOO Contd..

- Once a decision has been made to structure a shared memory as a collection of separate objects instead of as a linear address space, there are many other choices to be made.
- **Probably the most important issue is whether objects should be replicated or not. If replication is not used, all accesses to an object go through the one and only copy, which is simple, but may lead to poor performance.**
- **By allowing objects to migrate from machine to machine, as needed, it may be possible to reduce the performance loss by moving objects to where they are needed.**
- **On the other hand, if objects are replicated, what should be done when one copy is updated?**
- **One approach is to invalidate all the other copies, so that only the up-to-date copy remains. Additional copies can be created later, on demand, as needed.**
- An alternative choice is not to invalidate the copies, but to update them. Shared-variable DSM also has this choice, but for page-based DSM, invalidation is the only feasible choice.
- Similarly, object-based DSM, like shared-variable DSM, eliminates most false sharing.

Summary

- To summarize, object-based distributed shared memory offers three advantages over the other methods:

  1. It is more modular than the other techniques.

  2. The implementation is more flexible because accesses are controlled.

  3. Synchronization and access can be integrated together cleanly.
- Object-based DSM also has disadvantages. For one thing, it cannot be used to run old "dusty deck" multiprocessor programs that assume the existence of a shared linear address space that every process can read and write at random. However, since multiprocessors are relatively new, the existing stock of multiprocessor programs that anyone cares about is small.
- A second potential disadvantage is that since all accesses to shared objects must be done by invoking the objects' methods, extra overhead is incurred that is not present with shared pages that can be accessed directly.