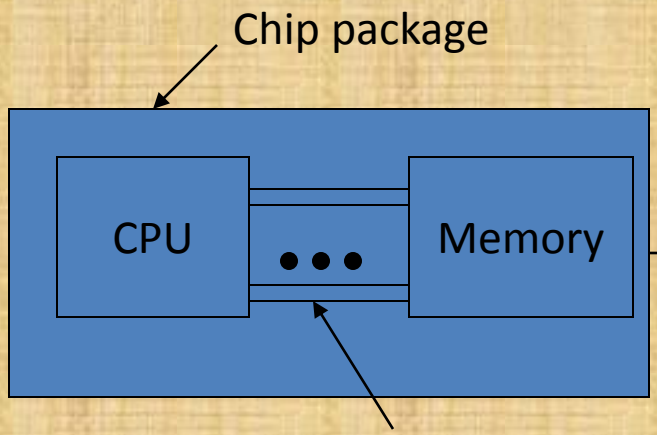
6. Distributed Shared Memory

What is shared memory?

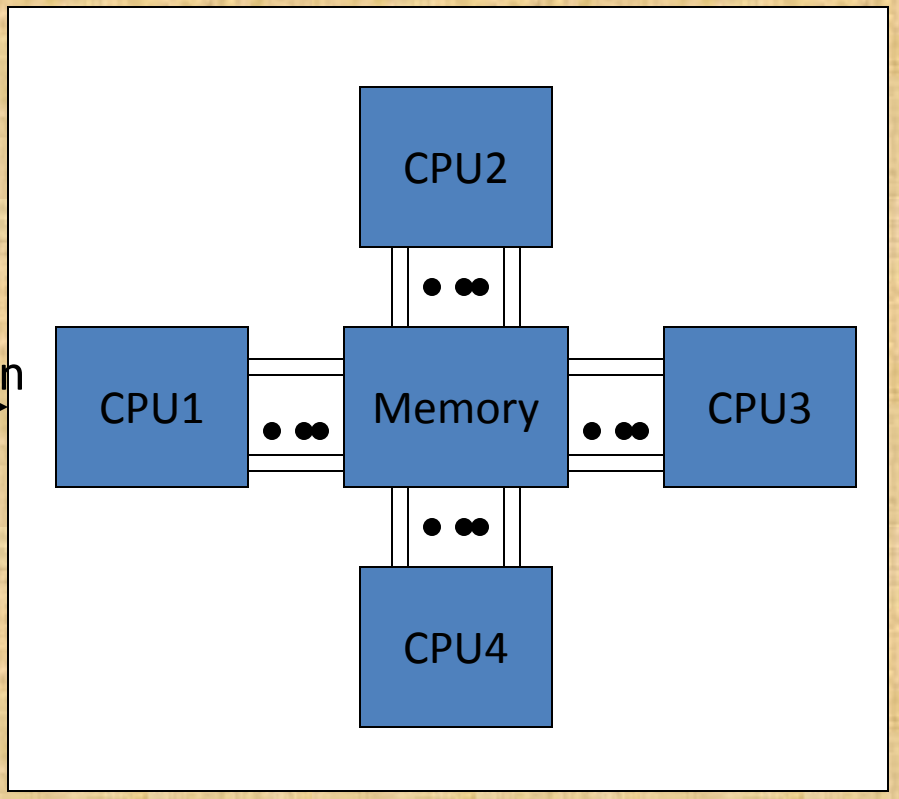
- **On-Chip Memory**



Address and data lines Connecting the CPU to the memory

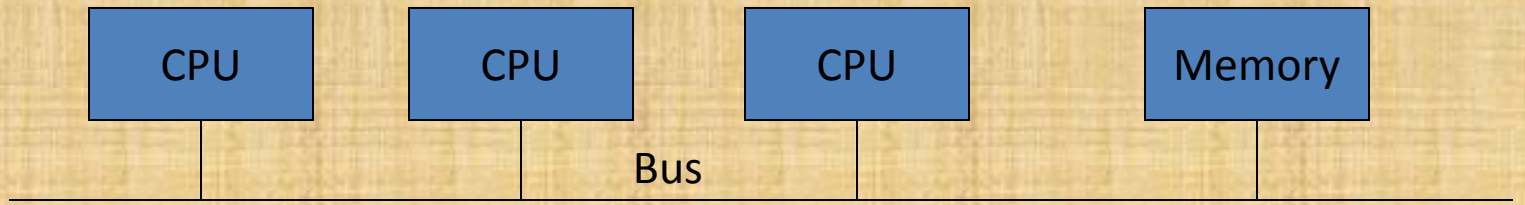
A single-chip computer

extension

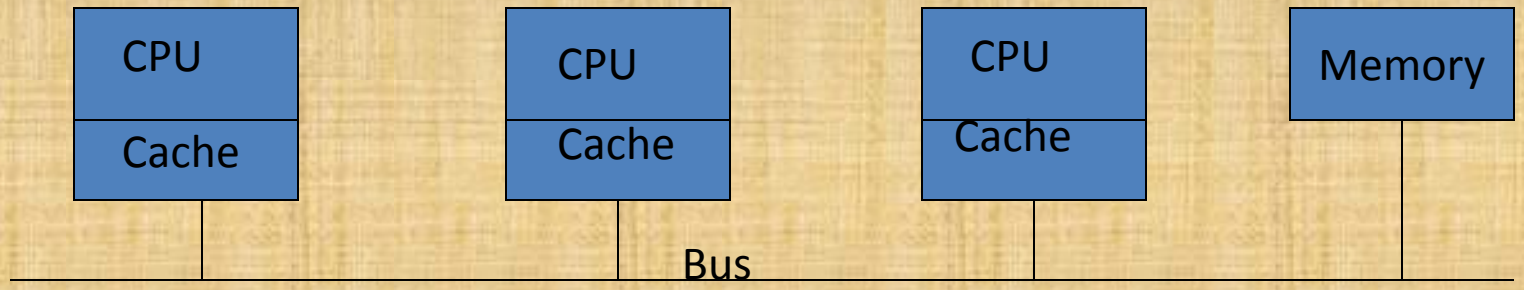


A hypothetical shared-memory Multiprocessor.

Bus-Based Multiprocessors



A multiprocessor



A multiprocessor with caching

Write through protocol

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

Write once protocol

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

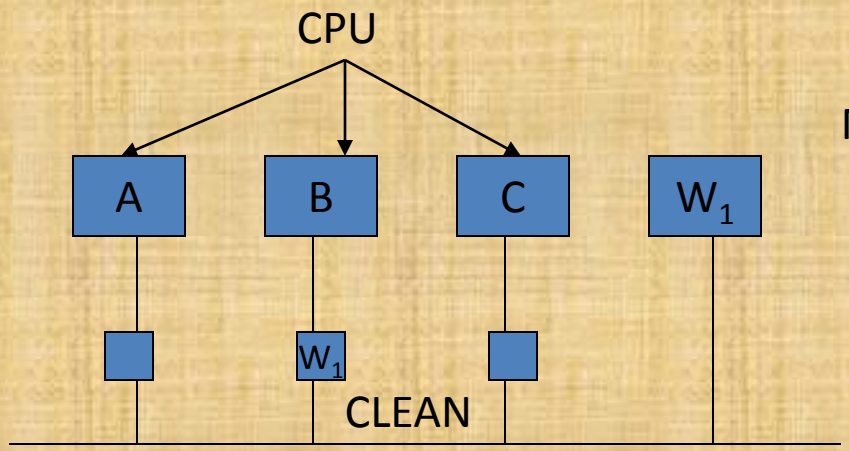
INVALID: This cache block does not contain valid data.

CLEAN: Memory is up-to-date; the block may be in other caches.

DIRTY: Memory is incorrect; no other cache holds the block.

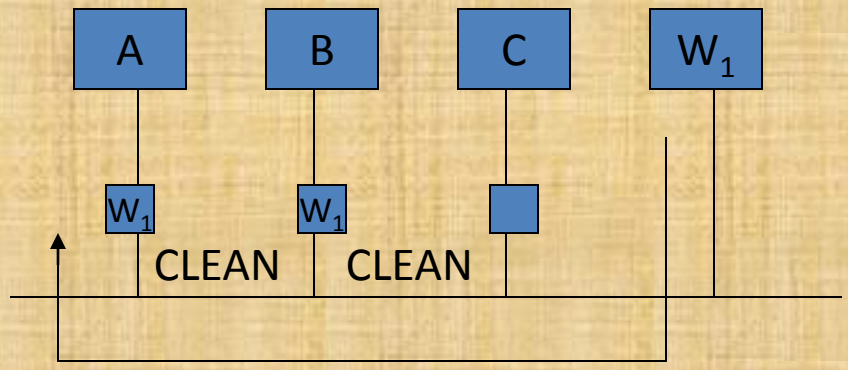
- The basic idea of the protocol is that a word that is being read by multiple CPUs is allowed to be present in all their caches. A word that is being heavily written by only one machine is kept in its cache and not written back to memory on every write to reduce bus traffic.

For example



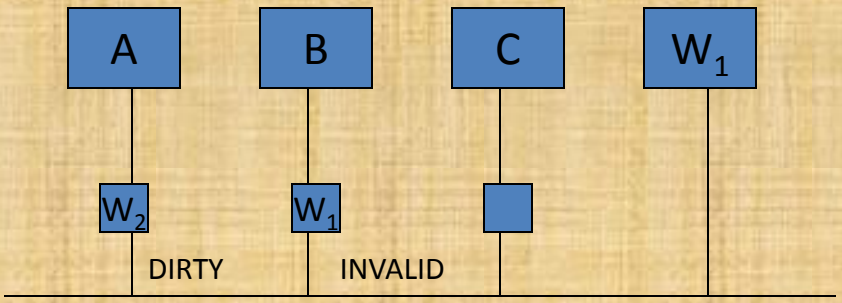
Memory is correct

(a) Initial state – word W_1 containing value W1 is in memory and is also cached by B.



Memory is correct

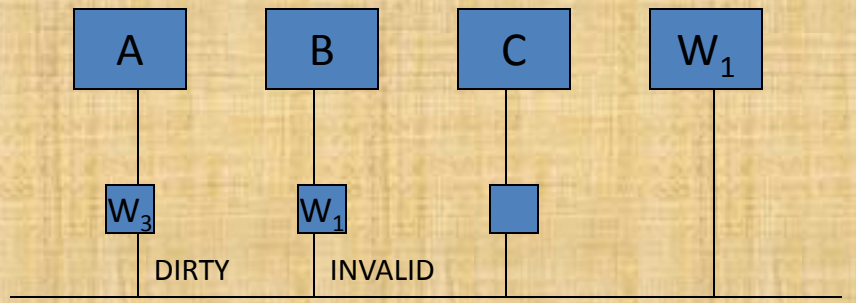
(b) A reades word W and gets W_1. B does not respond to the read, but the memory does.



Memory is correct

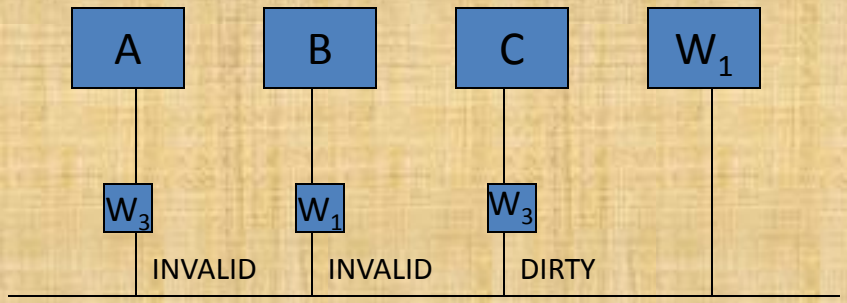
(c)A write a value W2, B snoops on the bus, sees the write, and invalidates its entry. A's copy is marked DIRTY.

Not update memory



Memory is correct

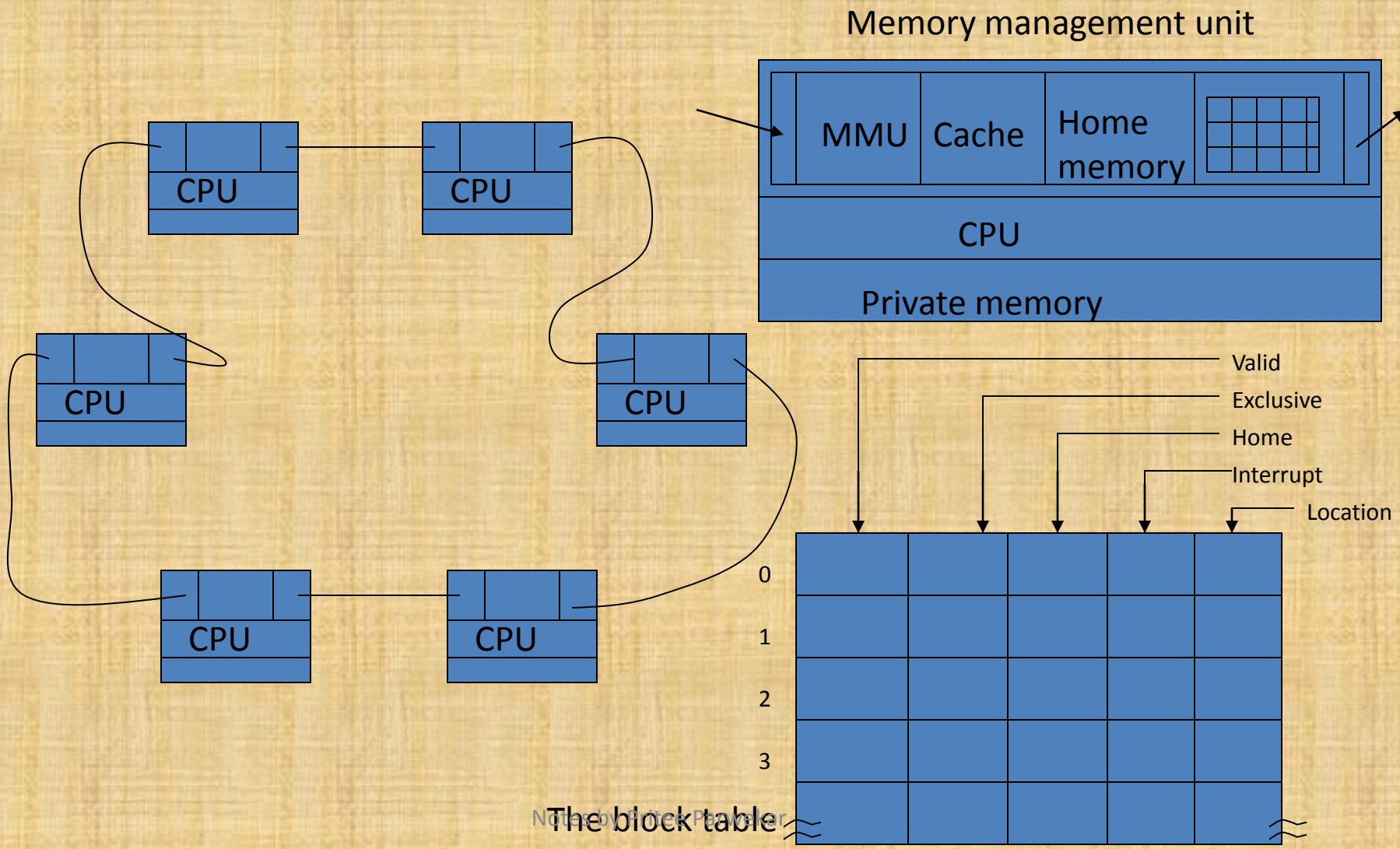
(d) A write W again. This and subsequent writes by A are done locally, without any bus traffic.



(e) 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.

Not update memory

Ring-Based Multiprocessors: Memnet



The block table

Notes by Mark Pavelec

Protocol

Read

When the CPU wants to read a word from shared memory, the memory address to be read is passed to the Memnet device, which checks the block table to see if the block is present. If so, the request is satisfied. If not, the Memnet device waits until it captures the circulating token, puts a request onto the ring. As the packet passes around the ring, each Memnet device along the way checks to see if it has the block needed. If so, it puts the block in the dummy field and modifies the packet header to inhibit subsequent machines from doing so.

If the requesting machine has no free space in its cache to hold the incoming block, to make space, it picks a cached block at random and sends it home. Blocks whose *Home* bit are set are never chosen because they are already home.

Write

If the block containing the word to be written is present and is the only copy in the system (i.e., the *Exclusive* bit is set), the word is just written locally.

If the needed block is present but it is not the only copy, an invalidation packet is first sent around the ring to force all other machines to discard their copies of the block about to be written. When the invalidation packet arrives back at the sender, the *Exclusive* bit is set for that block and the write proceeds locally.

If the block is not present, a packet is sent out that combines a read request and an invalidation request. The first machine that has the block copies it into the packet and discards its own copy. All subsequent machines just discard the block from their caches. When the packet comes back to the sender, it is stored there and written.

Switched Multiprocessors

Two approaches can be taken to attack the problem of not enough bandwidth:

Reduce the amount of communication. E.g. Caching.

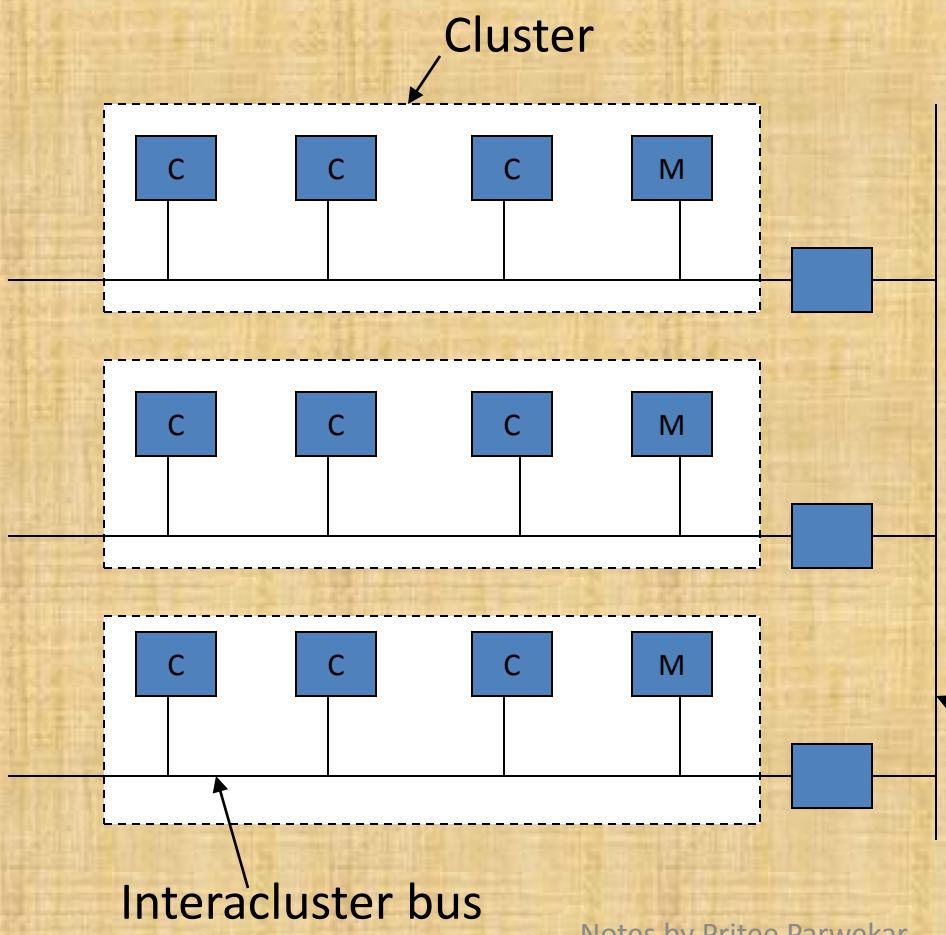
Increase the communication capacity. E.g. Changing topology.

One method is to build the system as a hierarchy. Build the system as multiple clusters and connect the clusters using an intercluster bus. As long as most CPUs communicate primarily within their own cluster, there will be relatively little intercluster traffic. If still more bandwidth is needed, collect a bus, tree, or grid of clusters together into a supercluster, and break the system into multiple superclusters.

Dash

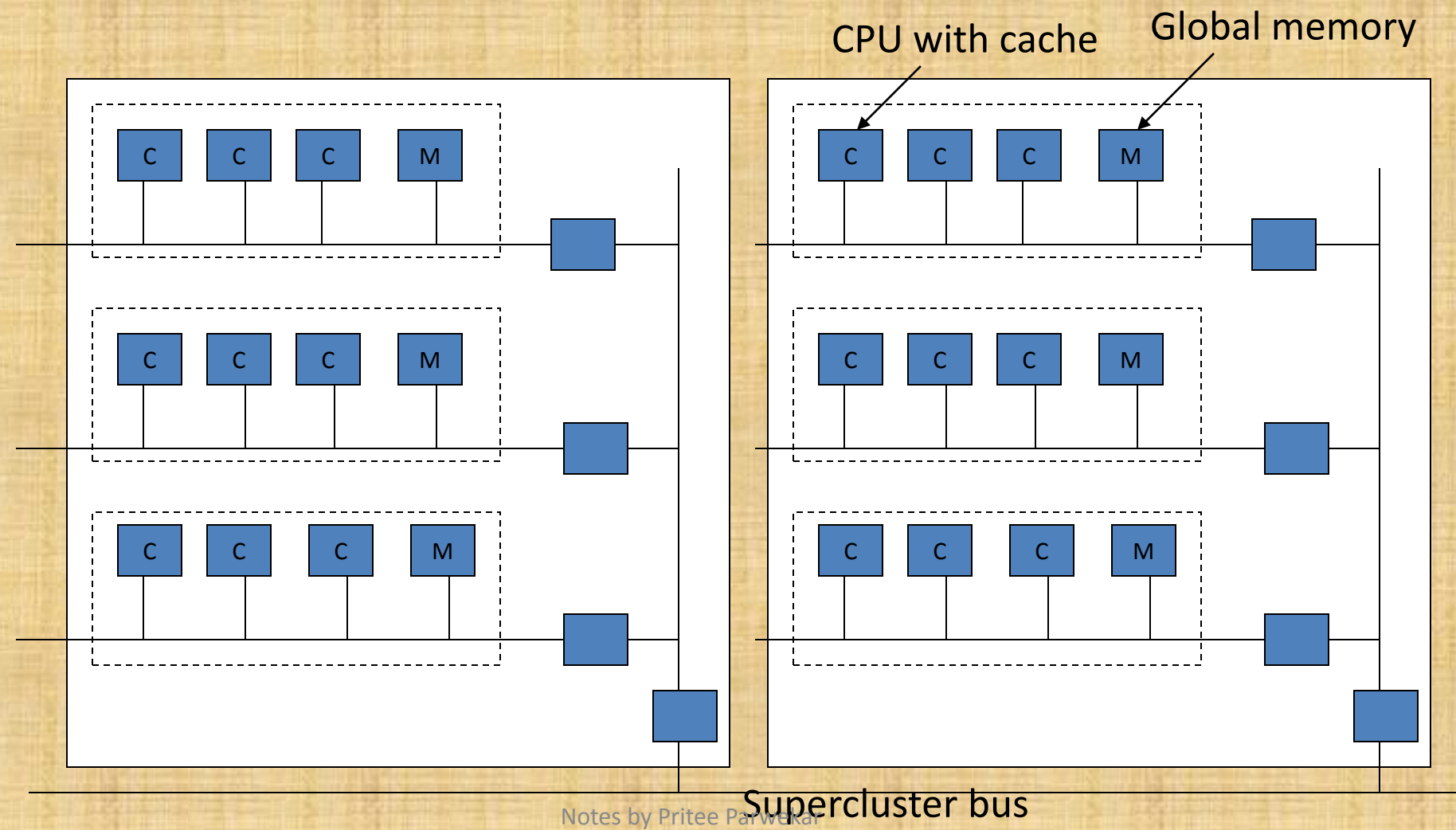
- A Dash consists of 16 clusters, each cluster containing a bus, four CPUs, 16M of the global memory, and some I/O equipment. Each CPU is able to snoop on its local bus, but not on other buses.
- The total address space is 256M, divided up into 16 regions of 16M each. The global memory of cluster 0 holds addresses 0 to 16M, and so on.

Intercluster Bus



Three clusters connected by an intercluster bus to form one supercluster.

Two superclusters connected by a supercluster bus



Notes by Pritee Parwekar

Caching

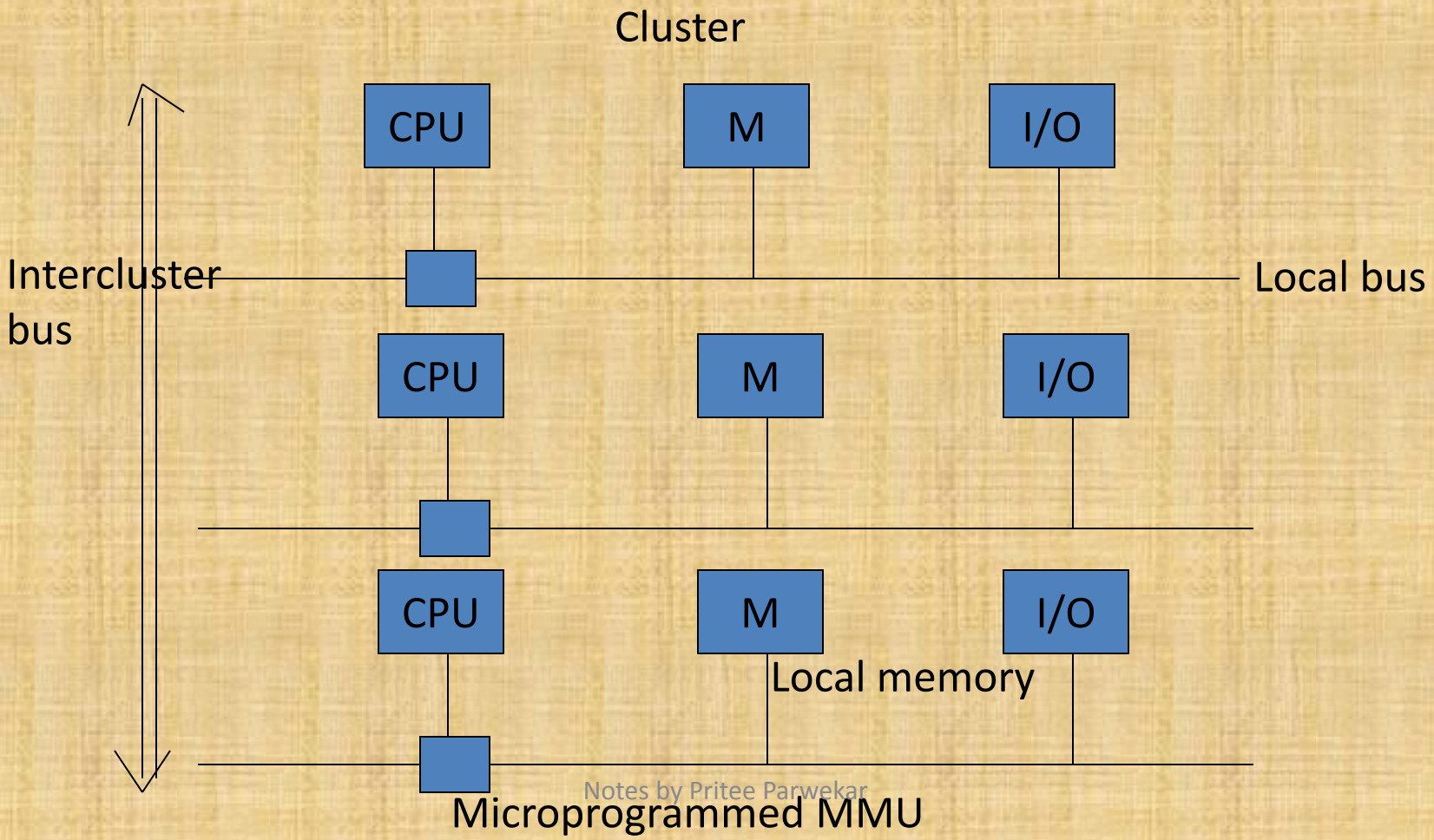
- Caching is done on two levels: a first-level cache and a larger second-level cache.
- Each cache block can be in one of the following three states:

- 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.

NUMA Multiprocessors

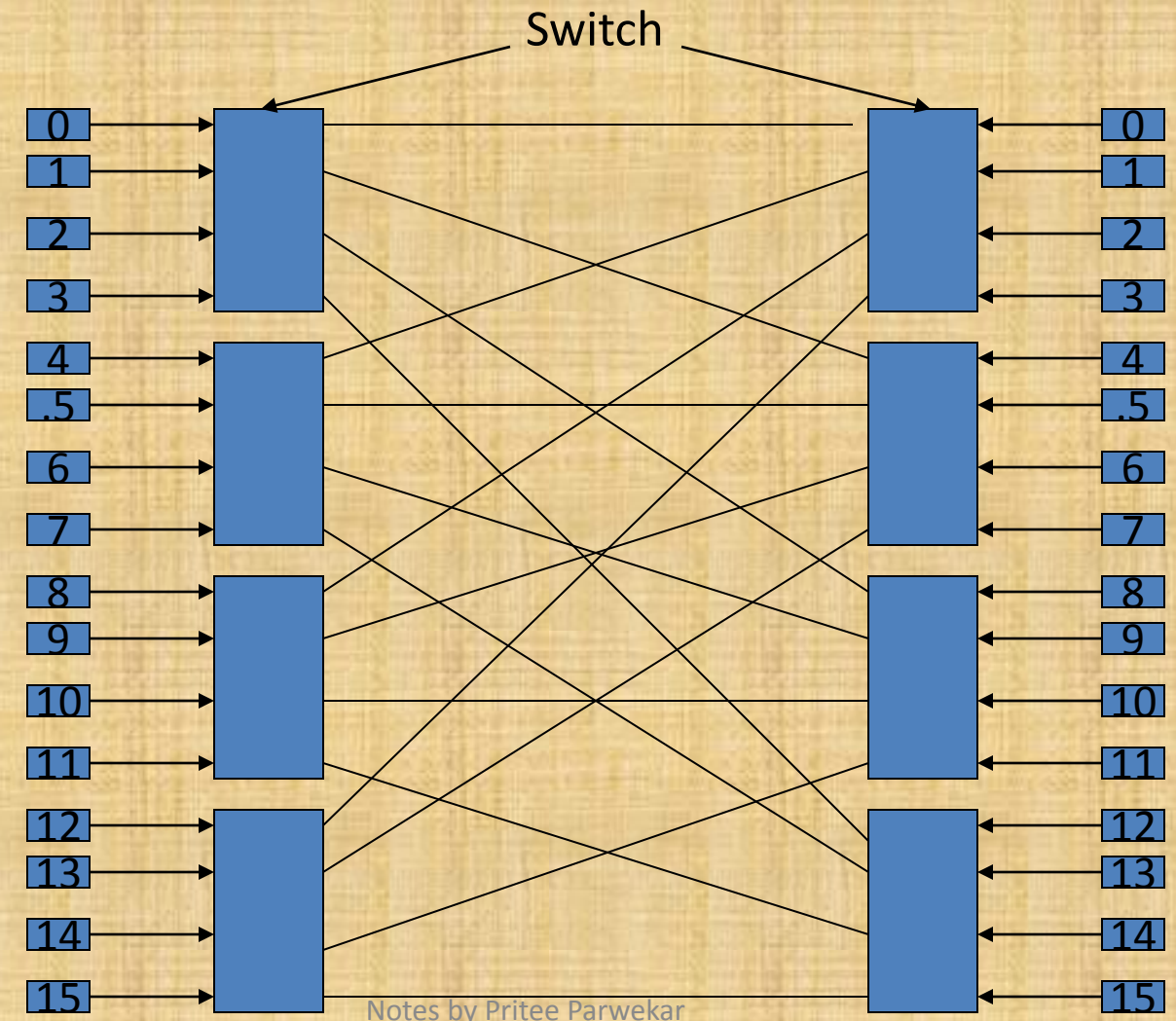
- Like a traditional **UMA** (**Uniform Memory Access**) multiprocessor, a NUMA machine has a single virtual address space that is visible to all CPUs. When any CPU writes a value to location *a*, a subsequent read of *a* by a different processor will return the value just written.
- The difference between UMA and NUMA machines lies in the fact that on a NUMA machine, access to a remote memory is much slower than access to a local memory.

Examples of NUMA Multiprocessors- Cm*



Notes by Pritee Parwekar

Examples of NUMA Multiprocessors- BBN Butterfly



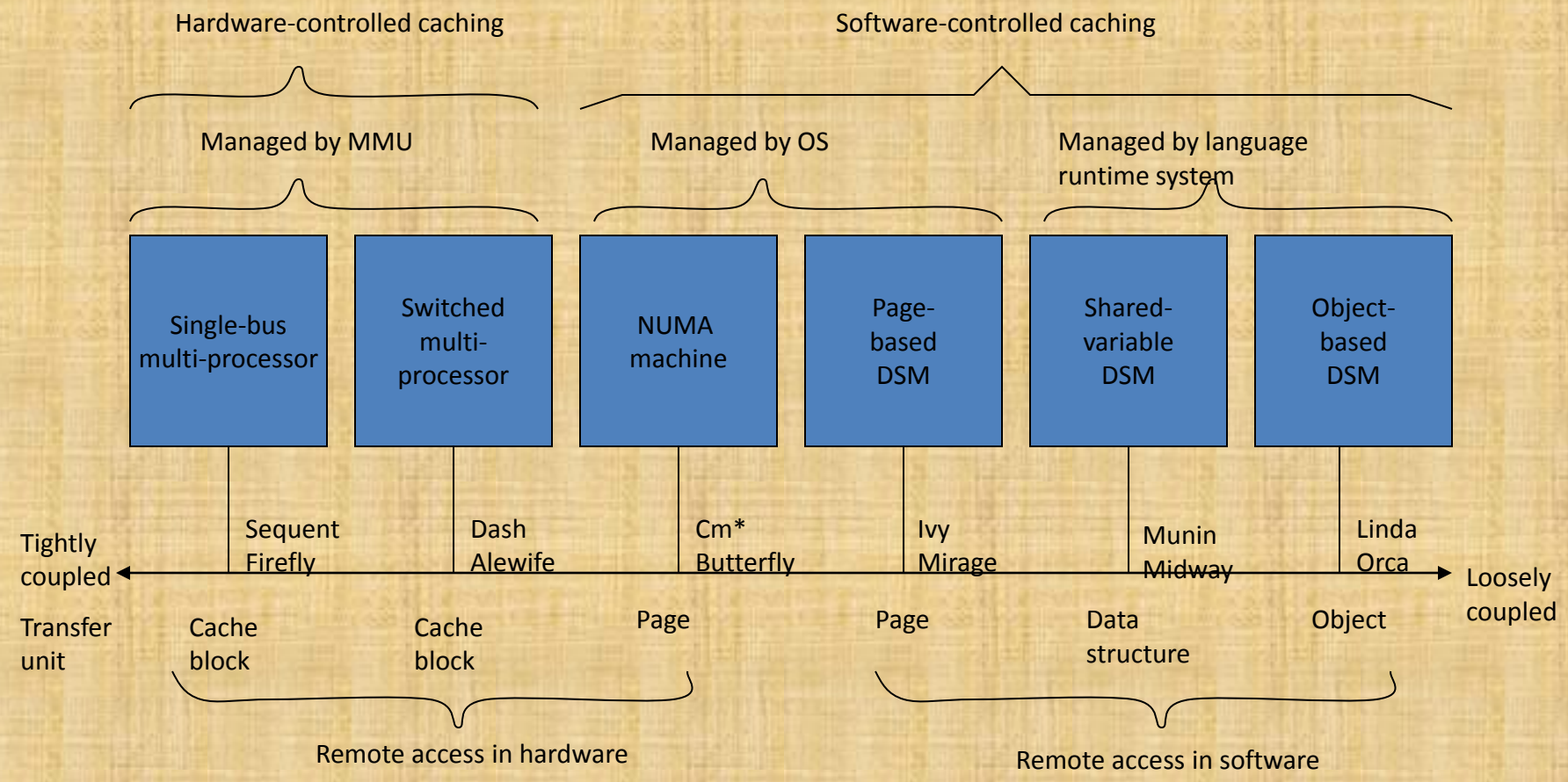
Properties of NUMA Multiprocessors

- Access to remote memory is possible
- Accessing remote memory is slower than accessing local memory
- Remote access times are not hidden by caching

NUMA algorithms

- In NUMA, it matters a great deal which page is located in which memory. The key issue in NUMA software is the decision of where to place each page to maximize performance.
- A **page scanner** running in the background can gather usage statistics about local and remote references. If the usage statistics indicate that a page is in the wrong place, the page scanner unmaps the page so that the next reference causes a page fault.

Comparison of Shared Memory Systems



	Multiprocessors			DSM		
Item	Single bus	Switched	NUMA	Page based	Shared variable	Object based
Linear, shared virtual address space?	Yes	Yes	Yes	Yes	No	No
Possible operations	R/W	R/W	R/W	R/W	R/W	General
Encapsulation and methods?	No	No	No	No	No	Yes
Is remote access possible in hardware?	Yes	Yes	Yes	No	No	No
Is unattached memory possible?	Yes	Yes	Yes	No	No	No
Who converts remote memory accesses to messages?	MMU	MMU	MMU	OS	Runtime system	Runtime system
Transfer medium	Bus	Bus	Bus	Network	Network	Network
Data migration done by	Hardware	Hardware	Software	Software	Software	Software
Transfer unit	Block	Block	Page	Page	Shared variable	Object

Notes by Pritee Parwekar

Consistency Models

Consistency Models

- **Strict Consistency**

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

P1: W(x)1

P2: R(x)1

Strict consistency

P1: W(x)1

P2: R(x)0 R(x)1

Not strict consistency

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.*

The following is correct:

P1: W(x)1 P1: W(x)1

P2: R(x)0 R(x)1 P2: R(x)1 R(x)1

Two possible results of running the same program

a=1; b=1; c=1;

print(b,c); print(a,c); print(a,b);

Three parallel processes: P1, P2, P3.

P1P2P3: 00 00 00 is not permitted.

P2P2P3: 00 10 01 is not permitted.

All processes see all shared accesses in the same order.

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.

P1:	W(x)1			W(x)3		
P2:		R(x)1	W(x)2			
P3:		R(x)1			R(x)3	R(x)2
P4:		R(x)1			R(x)2	R(x)3

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.*

P1: W(x)1

P2: R(x)1 W(x)2

P3: R(x)1 R(x)2

P4: R(x)2 R(x)1

A valid sequence of events for PRAM consistency but not for the above stronger models.

Processor consistency

- **Processor consistency** is PRAM consistency + memory coherence. That is, 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

The 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.*

P1: W(x)1 W(x)2 S

P2: R(x)1 R(x)2 S

P3: R(x)2 R(x)1 S

A valid sequence of events for weak consistency.

P1: W(x)1 W(x)2 S

P2: S R(x)1

An invalid sequence for weak consistency. P2 must get 2 instead of 1 because it is already synchronized.

Shared data can only be counted on to be consistent after a synchronization is done.

Release Consistency

- Release consistency provides **acquire** and **release** accesses. 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.

P1: Acq(L) W(x)1 W(x)2 Rel(L)

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

P3: R(x)1

A valid event sequence for release consistency. P3 does not acquire, so the result is not guaranteed.

- Shared data are made consistent when a critical region is exited.
- In **lazy release consistency**, at the time of a release, nothing is sent anywhere. Instead, when an acquire is done, the processor trying to do the acquire has to get the most recent values of the variables from the machine or machines holding them. Time stamp protocol is used.

Entry Consistency

- Shared data pertaining to a critical region are made consistent when a critical region is entered.
- Formally, a memory exhibits entry consistency if it meets all the following conditions:

1.The first condition says that when a process does an acquire the acquire may not complete until all the shared variables have been brought up to date.

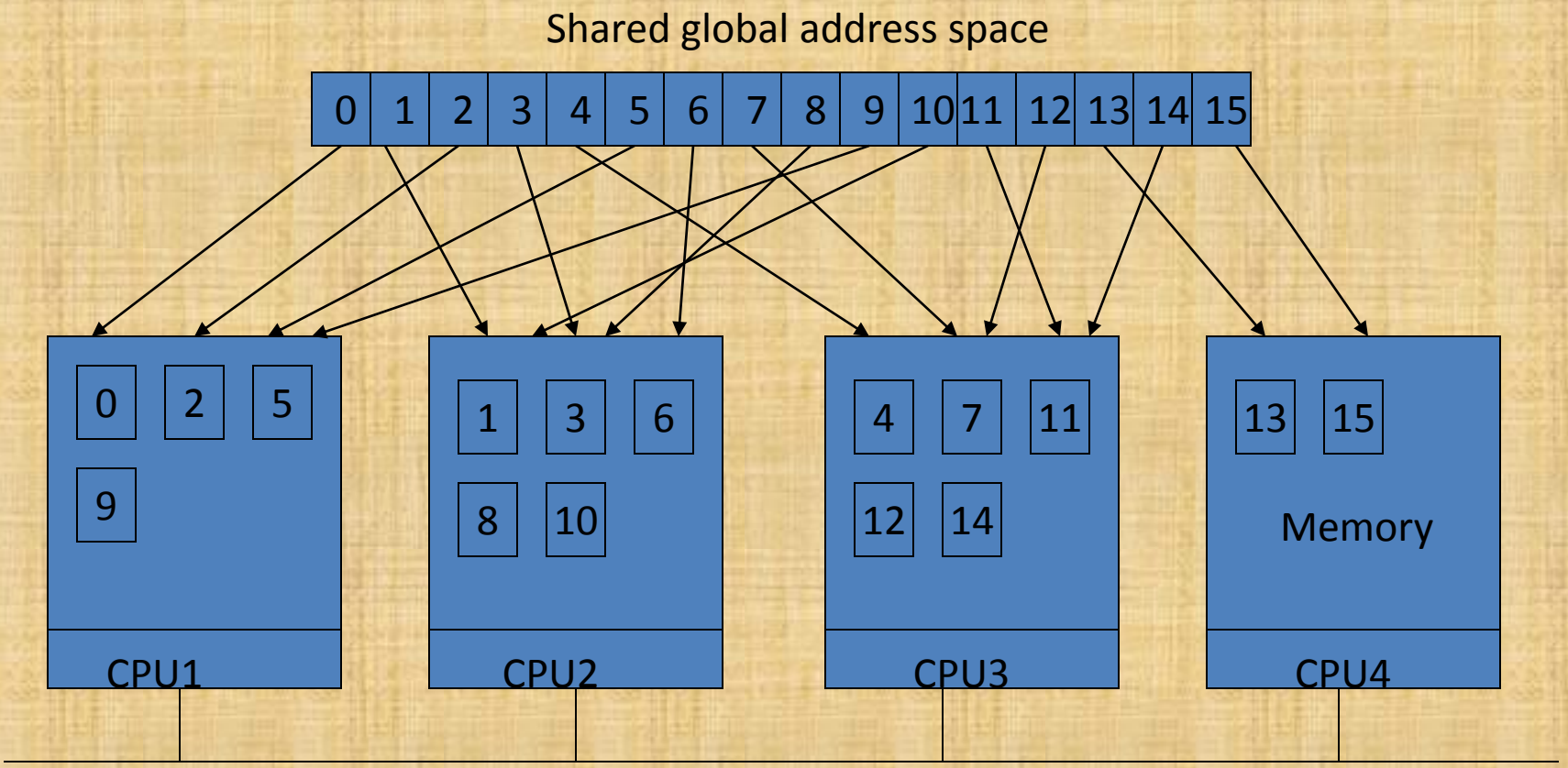
2.Second condition says that before updating a shared variable, a process must enter critical region in exclusive mode to make sure that no other process is trying to update it at the same time.

3. 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.

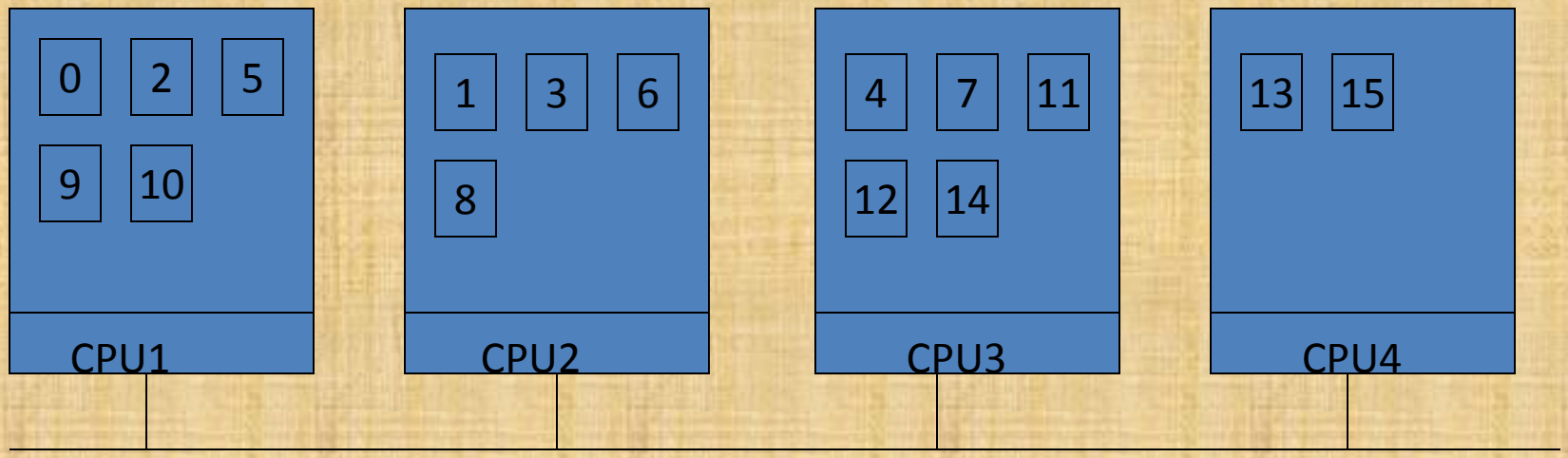
Page-based Distributed Shared Memory

- 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.

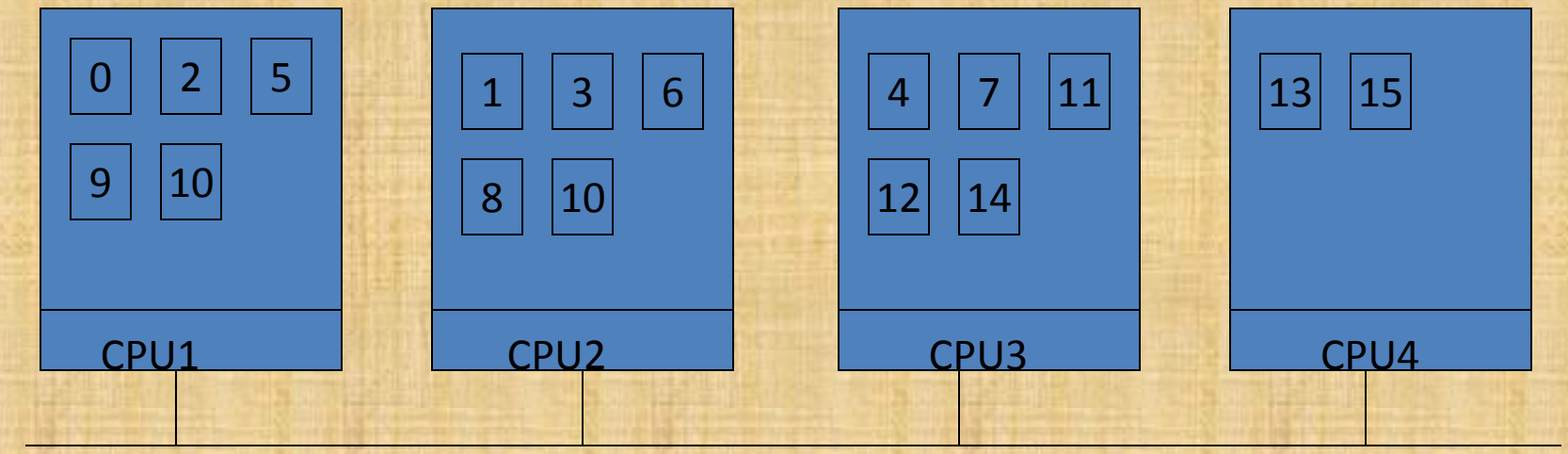
Chunks of address space distributed among four machines



Situation after CPU 1 references chunk 10



Situation if chunk 10 is read only and replication is used



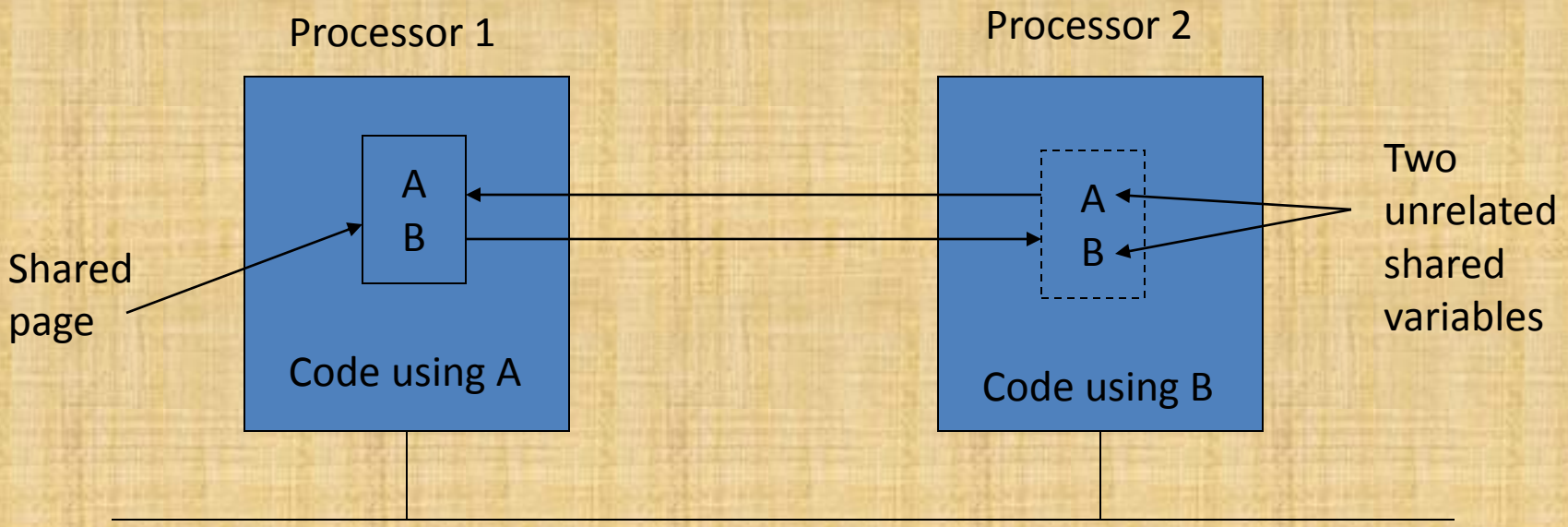
Replication

- One improvement to the basic system that can improve performance considerably is to replicate chunks that are read only.
- Another possibility is to replicate not only read-only chunks, but all chunks. No difference for reads, but if a replicated chunk is suddenly modified, special action has to be taken to prevent having multiple, inconsistent copies in existence.

Granularity

- when a nonlocal memory word is referenced, a chunk of memory containing the word is fetched from its current location and put on the machine making the reference. An important design issue is how big should the chunk be? A word, block, page, or segment (multiple pages).

False sharing



Finding the Owner

- The simplest solution is by doing a broadcast, asking for the owner of the specified page to respond.

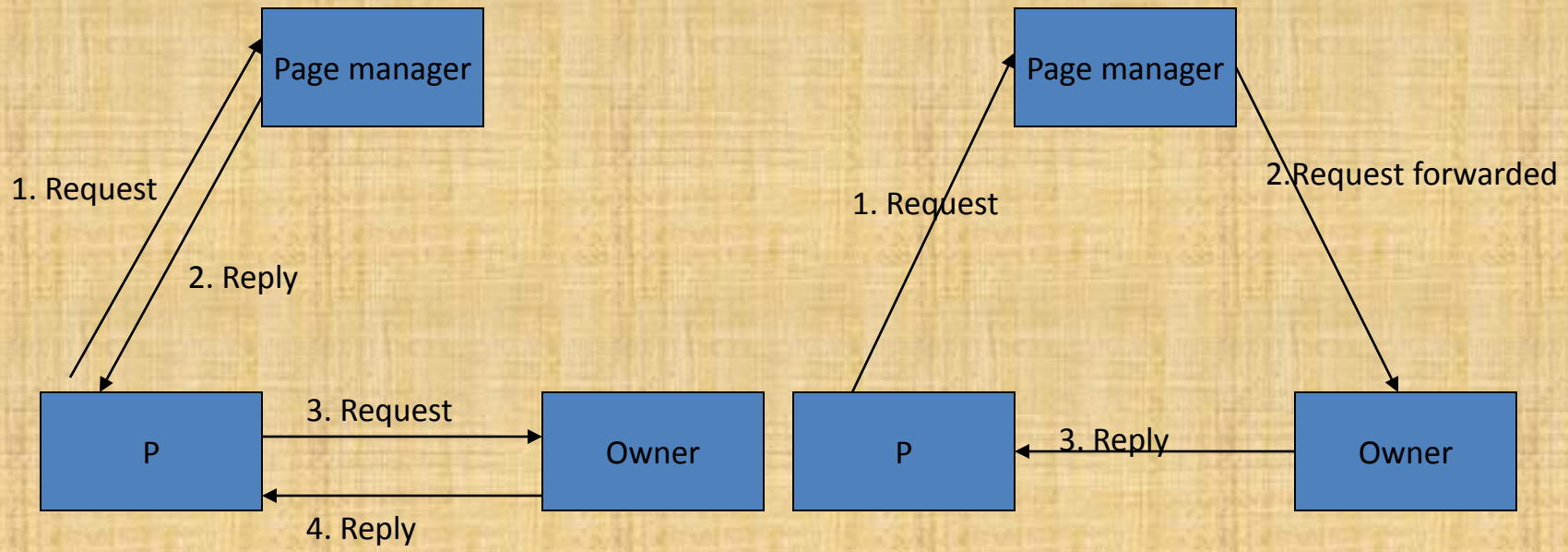
  Drawback: broadcasting wastes network bandwidth and interrupts each processor, forcing it to inspect the request packet.

- Another solution is to designate one process 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 or write a page it does not have, it sends a message to the page manager. The manager sends back a message telling who the owner is. Then P contacts the owner for the page.

- An optimization is to let manager forwards the request directly to the owner, which then replies directly back to P.

  Drawback: heavy load on page manager.

- Solution: having more page managers. Then how to find the right manager? One 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 is to use a hash function.


Page manager
1. Request
2. Reply
P
3. Request
Owner
4. Reply
Page manager
1. Request
2. Request forwarded
P
3. Reply
Owner

Finding the copies

- 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. 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.

Page Replacement

- If there is no free page frame in memory to hold the needed page, which page to evict and where to put it?
- using traditional virtual memory algorithms, such as LRU.
- In DSM, a replicated page that another process owns is always a prime candidate to evict because another copy exists.
- The second best choice is a replicated page that the evicting process owns. Pass the ownership to another process that owns a copy.
- If none of the above, then a nonreplicated page must be chosen. One possibility is to write it to disk. Another is to hand it off to another processor.

Synchronization

- In a DSM system, as in a multiprocessor, processes often need to synchronize their actions. A common example is mutual exclusion, in which only one process at a time may execute a certain part of the code. In a multiprocessor, the TEST-AND-SET-LOCK (TSL) instruction is often used to implement mutual exclusion. In a DSM system, this code is still correct, but is a potential performance disaster.

Shared-Variable Distributed Shared Memory

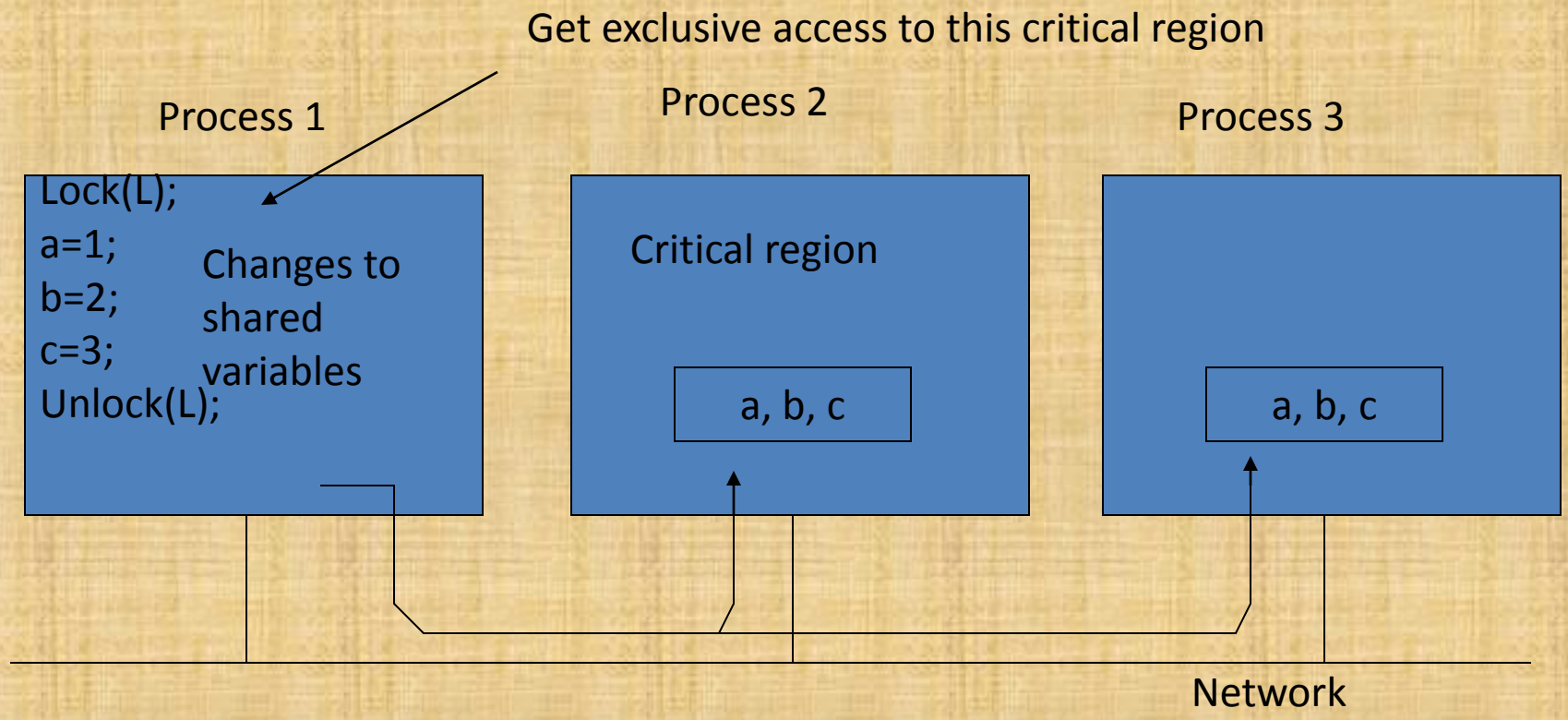
- Share only certain variables and data structures that are needed by more than one process.

- Two examples of such systems are Munin and Midway.

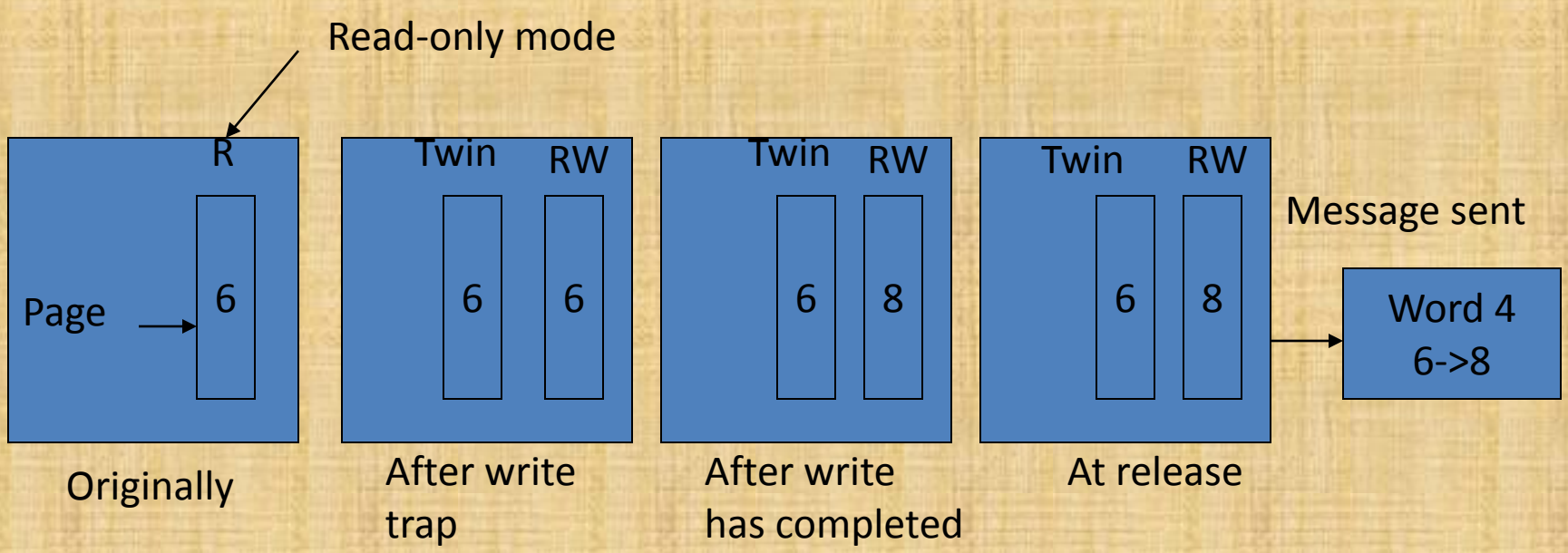
Munin

- Munin is a DSM system that is fundamentally based on software objects, but which can place each object on a separate page so the hardware MMU can be used for detecting accesses to shard objects.
- Is a software system for running shared memory parallel C++ programs on distributed memory multiprocessors
- Munin is based on a software implementation of release consisency.

Release consistency in Munin

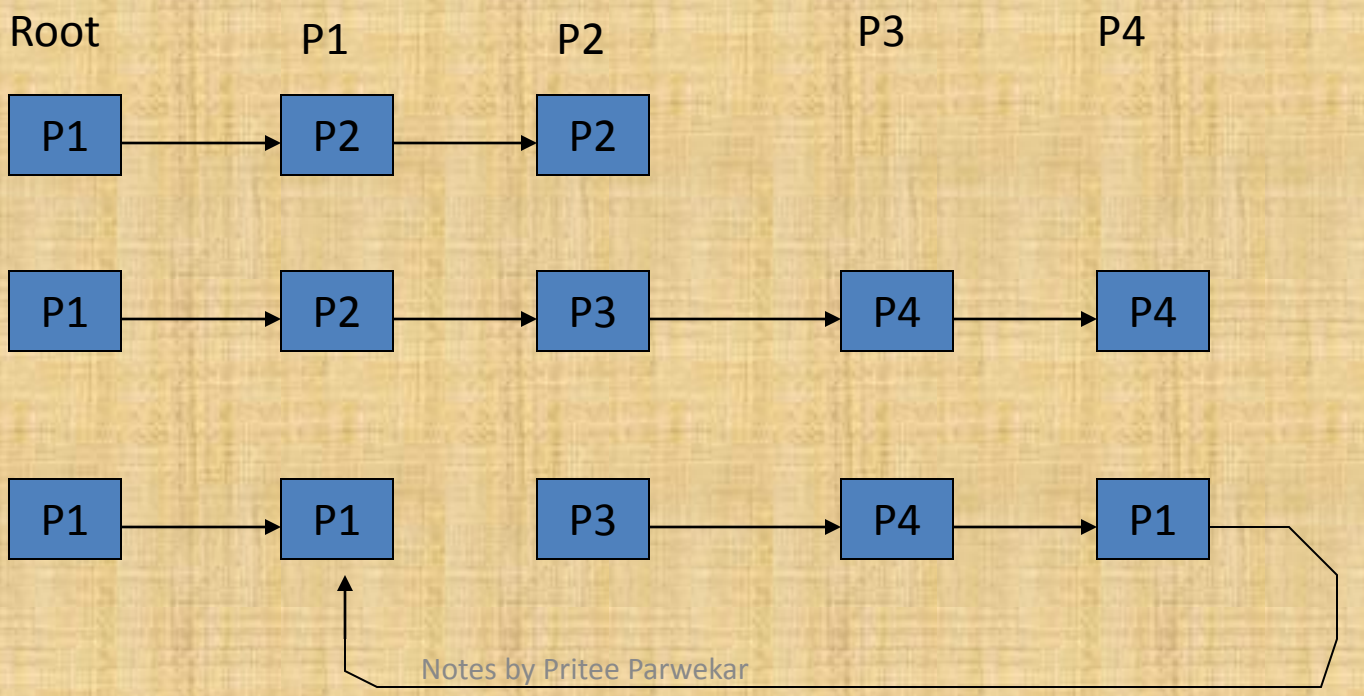


Use of twin pages in Munin



Directories

- Munin uses directories to locate pages containing shared variables.



Midway

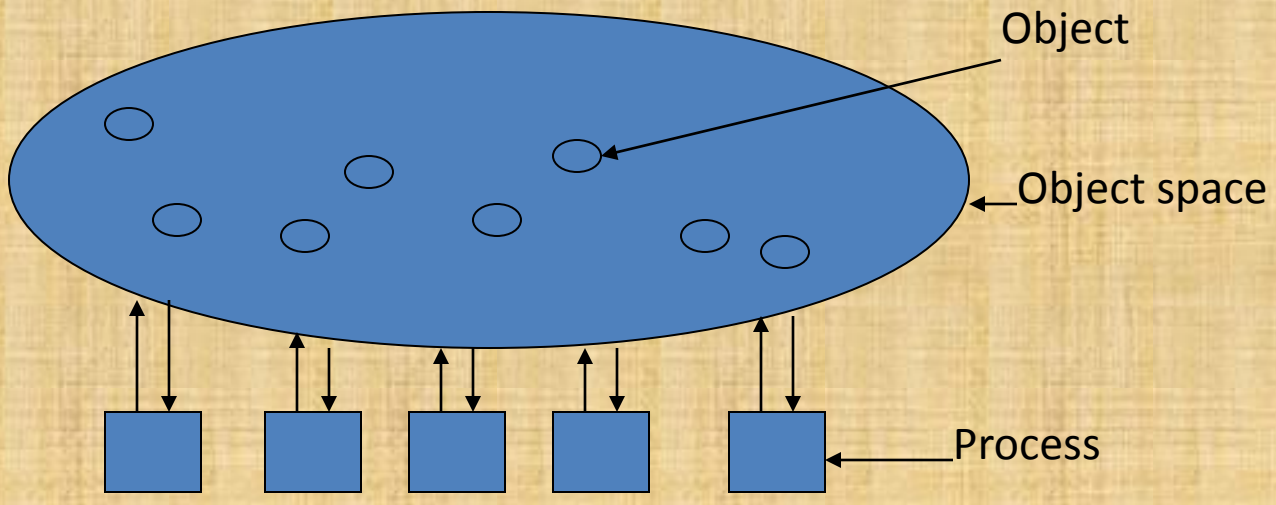
- Midway is a distributed shared memory system that is based on sharing individual data structures. It is similar to Munin in some ways, but has some interesting new features of its own.
- Midway supports entry consistency

Munin and Midway

- The consistency model they support, the number of consistency protocols used to implement the model, and implementation strategy.

Object-based Distributed Shared Memory

- In an object-based distributed shared memory, processes on multiple machines share an abstract space filled with shared objects.



Linda

- **Tuple Space**

A tuple is like a structure in C or a record in Pascal. It consists of one or more fields, each of which is a value of some type supported by the base language.

For example,

(“abc”,2,5)

(“matrix-1”,1,6,3.14)

(“family”,”is-sister”,”Carolyn”,”Elinor”)

- Operations on Tuples

Out, puts a tuple into the tuple space. E.g. out(“abc”,2,5);

In, retrieves tuple from the tuple space.

In(“abc”,2,?j);

If a match is found, j is assigned a value.

Orca

- Orca is a parallel programming system that allows processes on different machines to have controlled access to a distributed shared memory consisting of protected objects.
- These objects are a more powerful form of the Linda tuple, supporting arbitrary operations instead of just in and out.

A simplified stack object

```
Object implementation stack;
  top: integer;
  stack: array [integer 0..N-1] of integer;
  operation push (item: integer);
  begin
      stack [top] := item;
      top := top +1;
  end;
  operation pop ( ) : integer;
  begin
      guard top > 0 do
        top := top –1;
       return stack [top];
   od;
  end;
 begin
      top := 0;
 end;
```

- Orca has a fork statement to create a new process on a user-specified processor.

- e.g. for I in 1..n do fork foobar(s) on I; od;

End of Chapter 6