Distributed Shared Memory

- Shared Memory Systems
- Consistency Models
- Distributed Shared Memory Systems
  - page based
  - shared-variable based
- *Reading:*
  - *Coulouris: Distributed Systems, Addison Wesley, Chapter 17*
  - *A.S. Tanenbaum: Distributed Operating Systems, Prentice Hall, 1995, Chapter 6*
  - *M. Stumm and S. Zhou: Algorithms Implementing Distributed Shared Memory, IEEE Computer, vol 23, pp 54-64, May 1990*

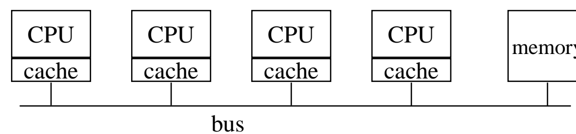
Distributed Shared Memory

- Shared memory: difficult to realize *vs*. easy to program with.
- Distributed Shared Memory (DSM): have collection of workstations share a single, virtual address space.
- Vanilla implementation:
  - references to local pages done in hardware.
  - references to remote page cause HW page fault; trap to OS; load the page from remote; restart faulting instruction.
- Optimizations:
  - share only selected portions of memory.
  - replicate shared variables on multiple machines.

Shared Memory

- DSM in context of shared memory for multiprocessors.
- Shared memory in multiprocessors:
  - On-chip memory
    - multiport memory, huh?
  - Bus-based multiprocessors
    - cache coherence
  - Ring-based multiprocessors
    - no centralized global memory.
  - Switched multiprocessors
    - directory based
  - NUMA (Non-Uniform Memory Access) multiprocessors
    - no attempt made to hide remote-memory access latency

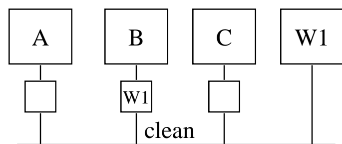
Bus-Based Multiprocessors



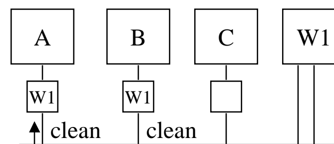
- snooping caches
- cache consistency protocols
- write through:

Event	*Action taken by a cache in response to its own CPU's operation*	*Action taken by a cache in response to remote CPU's operation*
Read miss	Fetch data from meory 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

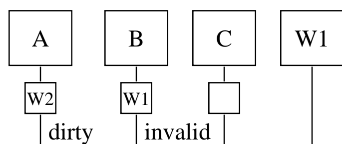
Bus-based Multiprocessors: Write-Once Cache Protocol



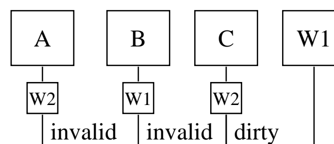
1. Initial state
Word with value W1 is in memory and also cached in B.



2. A reads word and gets W1.
B does not respond, but memory does.



3. A writes value W2. B invalidates entry.
A's copy is marked dirty.
Subsequent writes are done locally, without bus traffic.

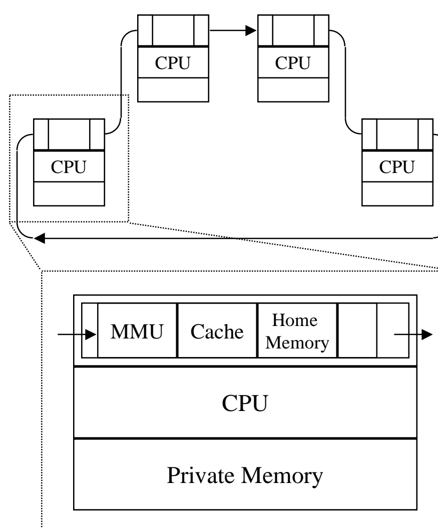


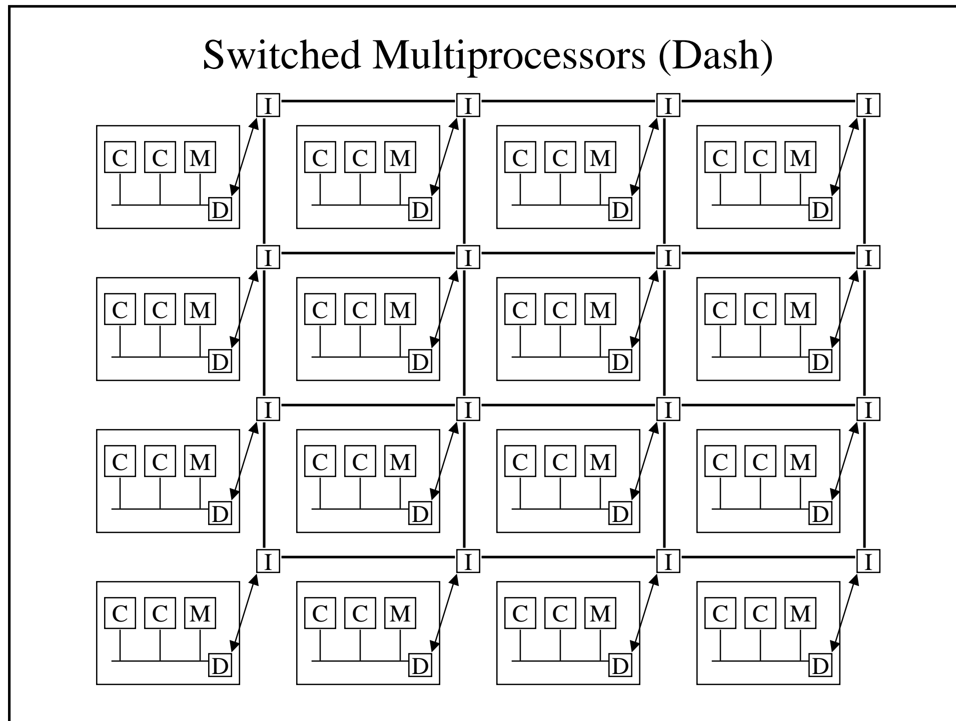
4. C reads or writes to word. A sees request, provides the value, and invalidates local entry. C now has only valid copy.

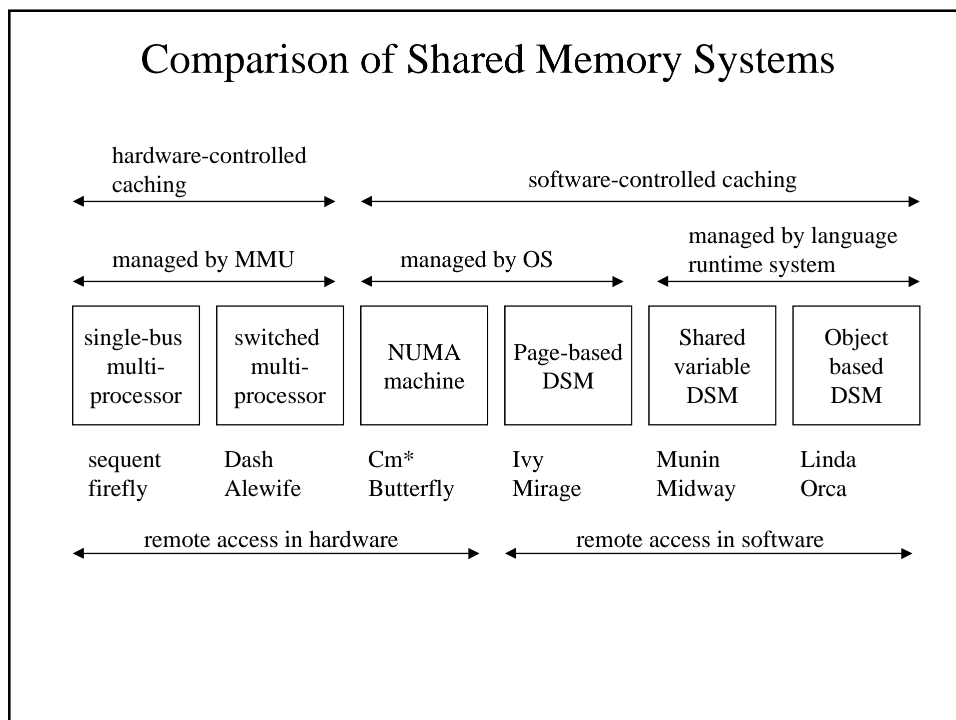
Ring-Based Multiprocessors

[Delp et al. 'Memory as a Network Abstraction", IEEE Network, vol 5, pp 34-41, July 1991]

- No centralized global memory.
- Memory blocks in shared space have *home memory field*.
- Read:
  1. wait for token
  2. send out request
  3. some machine has block; puts it in token, clear *exclusive* bit.
- Write:
  - block local; only copy
    - write locally
  - block is local; not only copy
    - send out invalidation packet
    - set exclusive field
  - block not local
    - send out request/invalidation




Switched Multiprocessors (Dash)
I
C
C
M
D


Comparison of Shared Memory Systems
hardware-controlled caching
software-controlled caching
managed by MMU
managed by OS
managed by language runtime system
single-bus multi-processor
switched multi-processor
NUMA machine
Page-based DSM
Shared variable DSM
Object based DSM
sequent firefly
Dash Alewife
Cm* Butterfly
Ivy Mirage
Munin Midway
Linda Orca
remote access in hardware
remote access in software

Consistency Models

- Single copy of writeable page:
  - simple, but expensive for heavily shared pages
- Multiple copies of writeable page
  - how to keep pages consistent?

- Perfect consistency is expensive.
- How to relax consistency requirements?

- Consistency model

Contract between application and memory.

If application agrees to obey certain rules, memory promises to work correctly.

Consistency Models

- Strict consistency
- Sequential consistency
- Causal consistency
- PRAM (pipeline RAM) consistency
- Weak consistency
- Release consistency

- increasing restrictions on application software
- increasing performance

Strict Consistency

- Most stringent consistency model:

  Any read to a memory location x returns the value stored by the most recent write operation to x.
- strict consistency observed in uni-processor systems.
- has come to be expected by uni-processor programmers
  - –very unlikely to be supported by any multiprocessor
- All writes are immediately visible by all processes
- Absolute global time order is maintained
- Two scenarios:

```
P1:  W(x)1
P2:           R(x)1

P1:  W(x)1
P2:          R(x)0  R(x)1
```

Sequential Consistency

- Strict consistency nearly impossible to implement.
- Programmers can manage with weaker models.
- Sequential consistency [Lamport 79]

  The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.
- Any valid interleaving is acceptable, but all processes must see the same sequence of memory references.
- Scenarios:

```
P1:  W(x)1
P2:          R(x)0  R(x)1

P1:  W(x)1
P2:          R(x)1  R(x)1
```

- Sequential consistency does not guarantee that read returns value written by another process anytime earlier.
- Results are not deterministic.

Sequential Consistency (cont)

- Implementation on DSM or multiprocessor
  - ensure that memory operation started only after all previous ones have been completed
  - example: use totally-ordered reliable broadcast mechanism to broadcast operations on shared variables
  - remember: exact order of operations does not matter as long as processes agree on the order of operations on the shared memory.
- Sequential consistency is programmer-friendly but expensive

Causal Consistency

- Weaken sequential consistency by making distinction between events that are potentially causally related and events that are not.
- USENET scenario: causality relations may be violated by propagation delays.
- 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.

- Scenario

```
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
```

Causal Consistency (cont)

- Other scenarios:

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

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

PRAM (pipelined RAM) Consistency

- Drop requirement that causally-related writes must be seen in same order by all machines.
- PRAM consistency:

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

- Scenario:

```
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
```

- Easy to implement: all writes generated by different processors are concurrent
- Counterintuitive result:

```
P1: a = 1;
    if (b==0) kill(P2);

P2: b = 1;
    if (a==0) kill(P1);
```

Weak Consistency

- PRAM consistency still unnecessarily restrictive for many applications: requires that writes originating in single process be seen everywhere in order.
- Example:
  - reading and writing of shared variables in tight loop inside a critical section.
  - Other processes are not supposed to touch variables, but writes are propagated to all memories anyway.
- Introduce synchronization variable:
  - When synchronization completes, all writes are propagated outward and all writes done by other machines are brought in.
  - All shared memory is synchronized.

Weak Consistency (cont)

1. Accesses to synchronization variables are sequentially ordered.

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.

- All processes see accesses to synchronization variables in same order.
- Accessing a synchronization variable "flushes the pipeline" by forcing writes to complete.
- By doing synchronization before reading shared data, a process can be sure of getting the most recent values.
- Scenarios:

```
P1: W(x)1 W(x)2   S
P2:                    R(x)1 R(x)2 S
P3:                    R(x)2 R(x)1 S
```

```
P1: W(x)1 W(x)2   S
P2:                    S  R(x)1
```

Release Consistency

- Problem with weak consistency:
  - When synchronization variable is accessed, we don't know if process is finished writing shared variables or about to start reading them.
  - Need to propagate all local writes to other machines and gather all writes from other machines.
- Operations:
  - `acquire` critical region: c.s. is about to be entered.
    - make sure that local copies of variables are made consistent with remote ones.
  - `release` critical region: c.s. has just been exited.
    - propagate shared variables to other machines.
  - Operations may apply to a subset of shared variables
- Scenario:

```
P1: Acq(L) W(x)1 W(x)2) Rel(L)
P2:                                Acq(L) R(x)2 Rel(L)
P3:                                                   R(x)1
```

Release Consistency (cont)

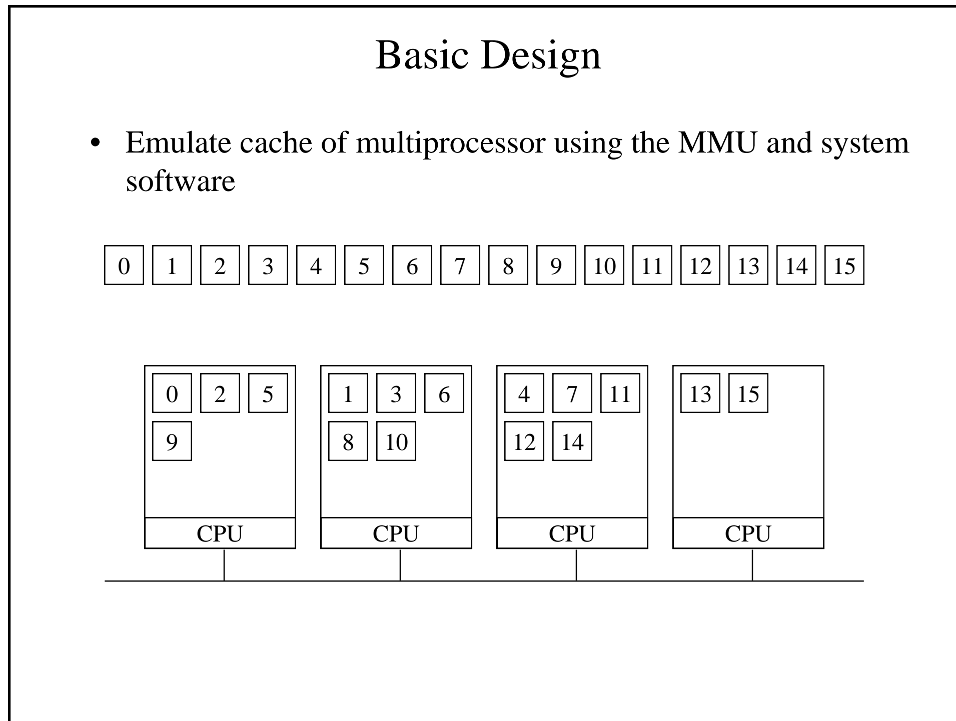
- Possible implementation
  - Acquire:
    1. Send request for lock to synchronization processor; wait until granted.
    2. Issue arbitrary read/writes to/from local copies.
  - Release:
    1. Send modified data to other machines.
    2. Wait for acknowledgements.
    3. Inform synchronization processor about release.
  - Operations on different locks happen independently.
- Release consistency:
  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 PRAM consistent (processor consistent) (sequential consistency is not required!)*

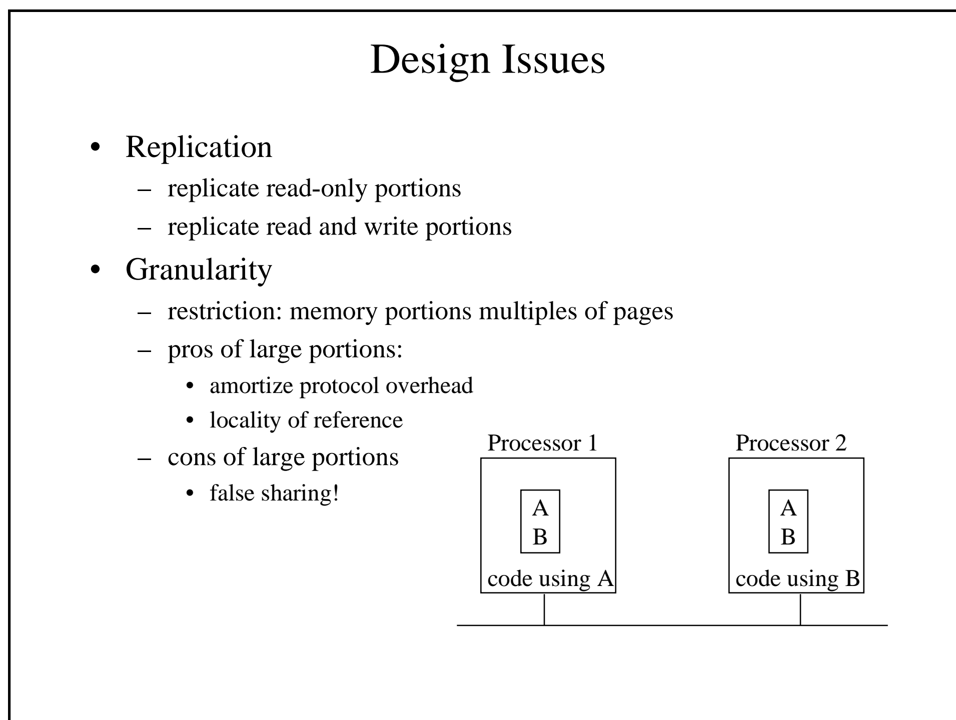
Consistency Models: Summary

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 causally-related shared accesses in the same order
PRAM	All processes see writes from each procesor in the order they were issued. Writes from different processors may not always be in the same order.
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

Page-Based DSM

- NUMA
  - processor can directly reference local and remote memory locations
  - no software intervention
- Workstations on network
  - can only reference local memory
- Goal of DSM
  - add software to allow NOWs to run multiprocessor code
  - simplicity of programming
  - “dusty deck” problem

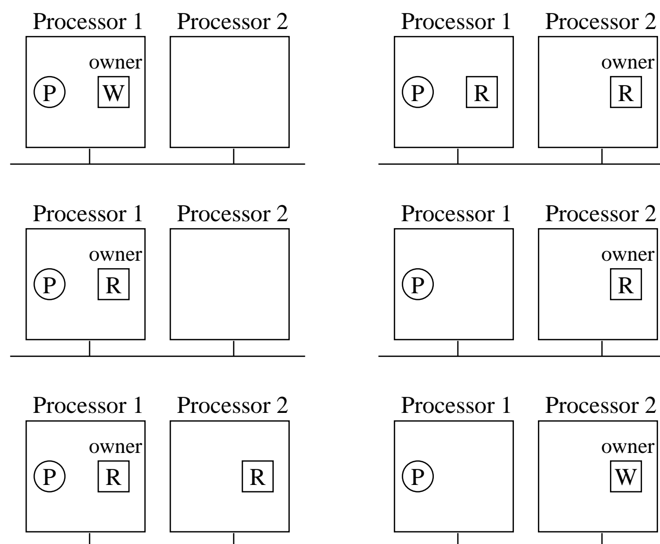

Basic Design
• Emulate cache of multiprocessor using the MMU and system software
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 2 5
9
CPU
1 3 6
8 10
CPU
4 7 11
12 14
CPU
13 15
CPU


Design Issues
• Replication
– replicate read-only portions
– replicate read and write portions
• Granularity
– restriction: memory portions multiples of pages
– pros of large portions:
• amortize protocol overhead
• locality of reference
– cons of large portions
• false sharing!
Processor 1
A
B
code using A
Processor 2
A
B
code using B

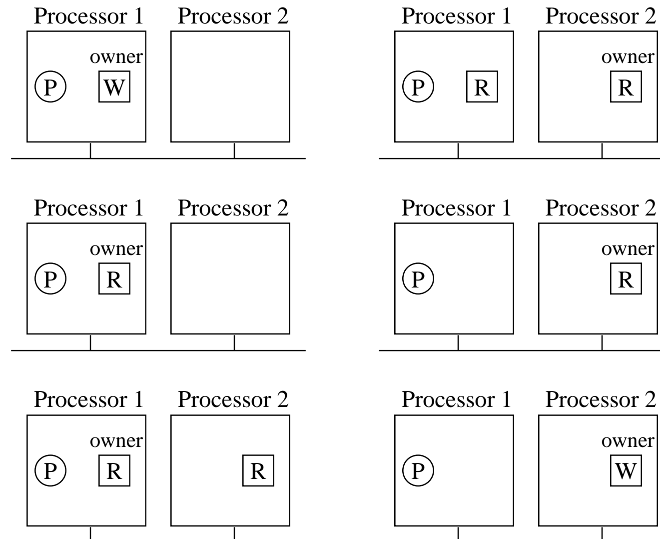
Design Issues (cont)

- Achieving Sequential Consistency
  - Only one copy of each page
    - consistency trivially guaranteed
  - Replicated pages:
    - read-only pages:
      - ok
    - read-write pages
      - read operation
        - install local copy, make it read-only
      - write operation
        - update or invalidate other copies?
        - invalidation typically used (why?)
    - Typical protocol
      - R (readable), W (readable and writeable) pages
      - each page has owner; process that most recently wrote to page

A Protocol for Sequential Consistency (reads)



A Protocol for Sequential Consistency (write)

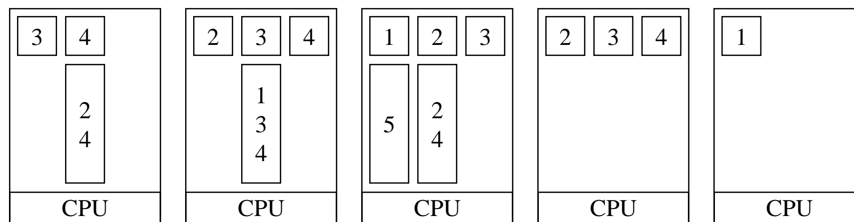


Design Issues (cont)

- Finding the Owner
  - broadcast request for owner
    - combine request with requested operation
    - problem: broadcast effects all participants (interrupts all processors), uses network bandwidth
  - page manager
    - possible hot spot
    - multiple page manager, hash on page address
  - probable owner
    - each process keeps track of probable owner
    - periodically refresh information about current owners

Design Issues (cont)

- Finding the copies
  - How to find the copies when they must be invalidated
  - broadcast requests
    - what when broadcasts are not reliable?
  - copysets
    - maintained by page manager or by owner



Design Issues (cont)

- Synchronization
  - locks
  - semaphores
  - barrier locks

  - Traditional synchronization mechanisms for multiprocessors don't work; why?
  - Synchronization managers

Shared-Variable DSM

- Is it necessary to share entire address space?
- Share individual variables.
- more variety in possible update algorithms for replicated variables
- opportunity to eliminate false sharing

- Examples: Munin (predecessor of Threadmarks)

[Bennet, Carter, Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", Proc Second ACM Symp. on Principles and Practice of Parallel Programming, ACM, pp. 168-176, 1990.]

Munin [Bennet *et al*, 1990]

- Use MMU: place each shared object onto separate page.

- Annotate declarations of shared variables.
  - keyword shared
  - compiler puts variable on separate page

- Synchronization:
  - lock variables
  - barriers
  - condition variables

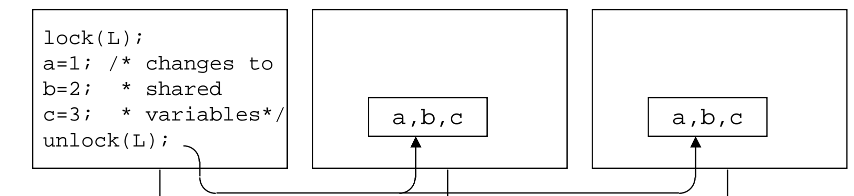
- Release consistency
- Multiple update protocols
- Directories for data location

Release Consistency in Munin

- Uses (eager) release consistency
- Critical regions
  - writes to shared variables occur inside critical region
  - reads can occur anywhere
  - when critical region exited, modified variables are brought up to date on all machines.
- Three classes of variables:
  - ordinary variable:
    - not shared; can be written only by process that created them.
  - shared data variables:
    - visible to multiple processes; appear sequentially consistent.
  - synchronization variable:
    - accessible via system-supplied access procedures
    - *lock/unlock* for locks, *increment/wait* for barriers

Release Consistency in Munin (cont)

- Example:

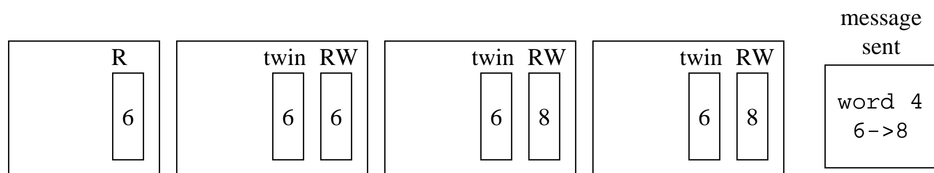


Multiple Protocols

- Annotations for shared-variable declarations:
  - read-only
    - do not change after initialization; no consistency problems
    - protected by MMU
  - migratory
    - not replicated; migrate from machine to machine as critical regions are entered
    - associated with a lock
  - write-shared
    - safe for multiple programs to write to it
    - use “diff” protocol to resolve multiple writes to same variable
  - conventional
    - treated as in conventional page-based DSM: only one copy of writeable page; moved between processors.

Twin Pages in Munin

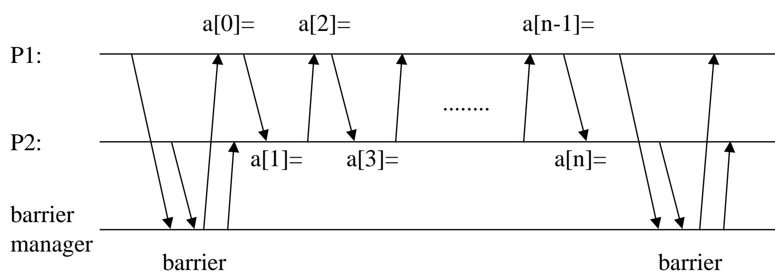
- Initially, write-shared page is marked as read-only.
- When write occurs, twin copy of page is made, and original page becomes read/write
- Release:
  - word-by-word comparison of dirty pages with their twins
  - send the differences to all processes needing them
  - reset page to read-only
  - compare incoming pages for modified words
  - if both local and incoming word have been modified, signal runtime error



Sequential *vs* Release Consistent Memory

Process1:
```
/* wait for process 2 */
wait_at_barrier(b);
for(i=0;i<n;i+=2)
  a[i] = a[i]+f(i);
/* wait until proc 2 is done */
wait_at_barrier(b);
```

Process2:
```
/* wait for process 1 */
wait_at_barrier(b);
for(i=1;i<n;i+=2)
  a[i] = a[i]+g(i);
/* wait until proc 1 is done */
wait_at_barrier(b);
```



Sequential *vs* Release Consistent Memory

Process1:
```
/* wait for process 2 */
wait_at_barrier(b);
for(i=0;i<n;i+=2)
  a[i] = a[i]+f(i);
/* wait until proc 2 is done */
wait_at_barrier(b);
```