CSCE 613: Interlude: Distributed Shared Memory

- Shared Memory Systems
- Consistency Models
- Distributed Shared Memory Systems
  - page based
  - shared-variable based
- *Reading (old!):*
  - *Coulouris: Distributed Systems, Addison Wesley, Chapter 17*
  - *Tanenbaum: Distributed Operating Systems, Prentice Hall, 1995, Chapter 6*
  - *Tanenbaum, van Steen: Distributed Systems, Prentice Hall, 2002, Chapter 6.2*
  - *M. Stumm and S. Zhou: Algorithms Implementing Distributed Shared Memory, IEEE Computer, vol 23, pp 54-64, May 1990*

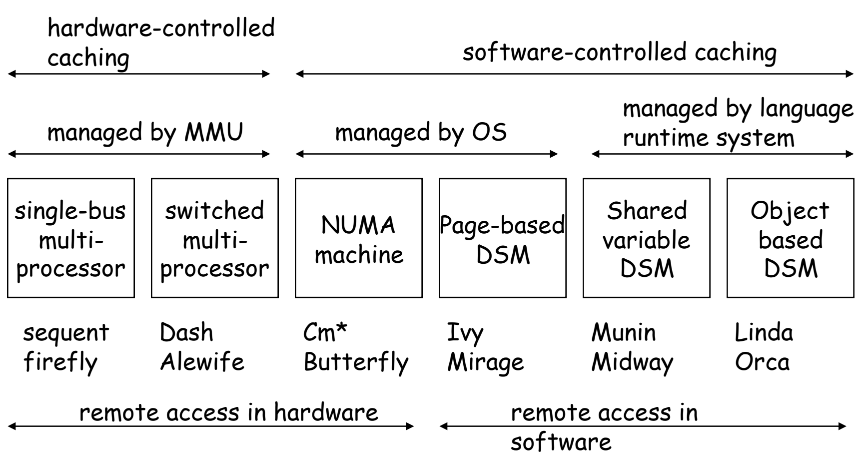
Distributed Shared Memory

- **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

Comparison of (old!) Shared Memory Systems



Prologue for DSM: Memory Consistency Models

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

- Definition: **Consistency Model:** Contract between application and memory. If application agrees to obey certain rules, memory promises to work correctly.

Memory Consistency: Example

- Example: Critical Section

```
/* lock(mutex) */
< implementation of lock would come here>
/* counter++ */
load  r1, counter
add   r1, r1, 1
store r1, counter
/* unlock(mutex) */
store zero, mutex
```

- Relies on all CPUs seeing update of **counter** before update of **mutex**
- Depends on assumptions about ordering of stores to memory

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 simple 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
- Requires that absolute global time order is maintained

- Two scenarios:

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

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

Example of Strong Ordering: Sequential Ordering

- **Strict Consistency** is impossible to implement.
- **Sequential Consistency**:
  - Loads and stores execute in program order
  - Memory accesses of different CPUs are "sequentialised"; i.e., any valid interleaving is acceptable, but all processes must see the same sequence of memory references.

- Traditionally used by many architectures

CPU 0	CPU 1
`store r1, adr1`	`store r1, adr2`
`load r2, adr2`	`load r2, adr1`

- In this example, at least one CPU must load the other's new value.

Sequential Consistency

- Strict consistency 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 <u>some</u> sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.
- Memory accesses of different CPUs are "sequentialised"; Any valid interleaving is acceptable, but all processes must see the same sequence of memory references.

- Scenarios:

P1:	W(x)1				
P2:		W(x)0			
P3:			R(x)0		R(x)1
P4:				R(x)0	R(x)1

P1:	W(x)1				
P2:		W(x)0			
P3:			R(x)0		R(x)1
P4:				R(x)1	R(x)0

Sequential Consistency: Observations

- Sequential consistency does not guarantee that read returns value written by another process anytime earlier.
- Results are not deterministic.
- Sequential consistency is programmer-friendly, but expensive.
- Lipton & Sandbert (1988) show that improving the read performance makes write performance worse, and vice versa.
- Modern HW features interfere with sequential consistency; e.g.:
  - write buffers to memory (aka store buffer, write-behind buffer, store pipeline)
  - instruction reordering by optimizing compilers
  - superscalar execution
  - pipelining

Linearizability (Herlihy and Wing, 1991)

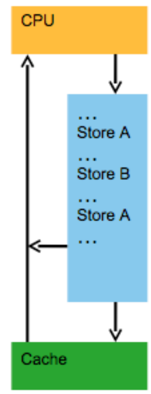
- Assume that events are timestamped with clock with finite precision (e.g.loosely synchronized clocks).
- Let $ts_{OP}(x)$ be timestamp of operation *OP* on data item *x*. *OP* is either a **`read(x)`** or a **`write(x)`**.

  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. In addition, if $ts_{OP1}(x) < ts_{OP2}(x)$, then operation OP1(x) should precede OP2(x) in this sequence.

- Stricter than Sequential Consistency.

Weaker Consistency Models: **Total Store Order**

- **Total Store Ordering** (TSO) guarantees that the sequence in which `store`, `FLUSH`, and `atomic load-store` instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor.
- Both x86 and SPARC processors support TSO.
- A later `load` can bypass an earlier `store` operation. (!)
- i.e., local `load` operations are permitted to obtain values from the write buffer before they have been committed to memory.

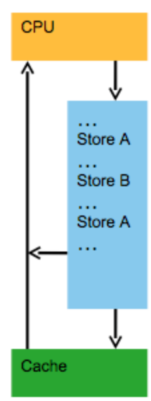


Total Store Order (cont)

- Example:

CPU 0	CPU 1
`store r1, adr1`	`store r1, adr2`
`load r2, adr2`	`load r2, adr1`

- Both CPUs may read old value!
- Need hardware support to force global ordering of privileged instructions, such as:
  - atomic swap
  - test & set
  - load-linked + store-conditional
  - memory barriers
- For such instructions, stall pipeline and flush write buffer.



It gets weirder: **Partial Store Ordering**

- **Partial Store Ordering** (PSO) does **not** guarantee that the sequence in which `store`, `FLUSH`, and `atomic load-store` instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor.
- The processor can reorder the stores so that the sequence of stores to memory is not the same as the sequence of stores issued by the CPU.
- SPARC processors support PSO; x86 processors do not.
- Ordering of stores is enforced by **memory barrier** (instruction `STBAR` for Sparc) : If two stores are separated by memory barrier in the issuing order of a processor, or if the instructions reference the same location, the memory order of the two instructions is the same as the issuing order.

Partial Store Order (cont)

- Example:

```
/* lock(mutex) */
< implementation of lock would come here>
/* counter++ */
load  r1, counter
add   r1, r1, 1
store r1, counter
/* MEMORY BARRIER */
STBAR
/* unlock(mutex) */
store zero, mutex
```

- Store to `mutex` can "overtake" store to `counter`.
- Need to use **memory barrier** to separate issuing order.
- Otherwise, we have a race condition.

Causal Consistency

- Weaken sequential consistency by making distinction between events that are potentially causally related and events that are not.
- Distributed forum 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 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.

- 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

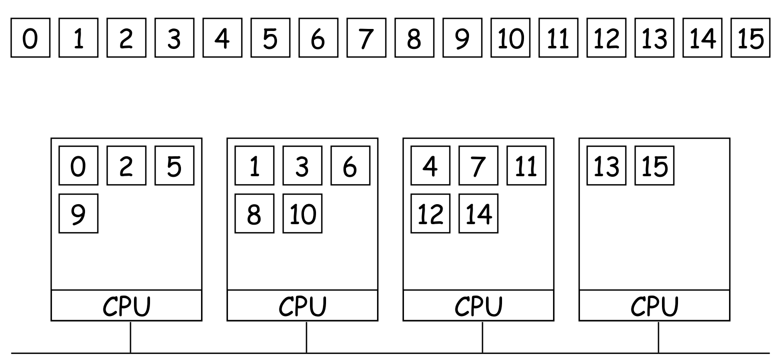
Consistency	Description
Strict	Absolute time ordering of all shared accesses matters
Sequential	All processes see all shared accesses in the same order
Causal	All processes see all causally-related shared accesses in the same order
PRAM	All processes see writes from each procesor in the order they were 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

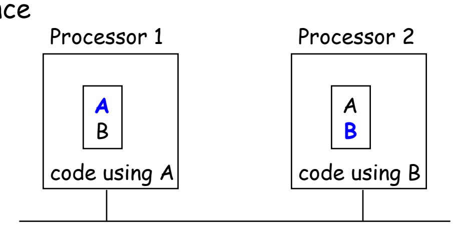
Basic Design

- Emulate cache of multiprocessor using the MMU and system software



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!

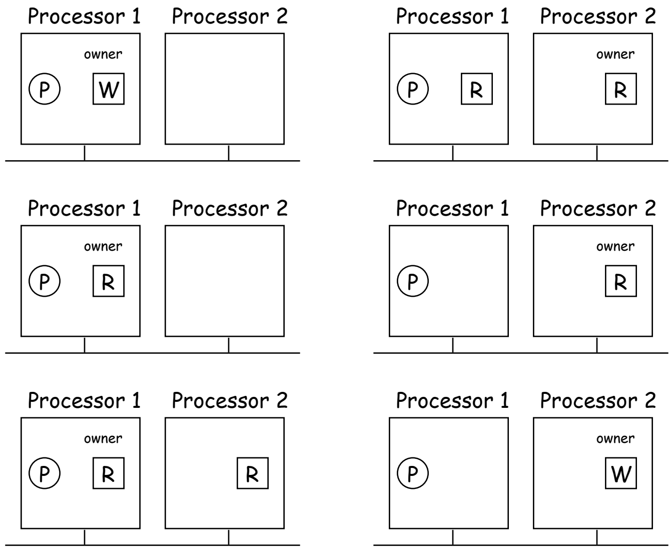


Design Issues (cont)

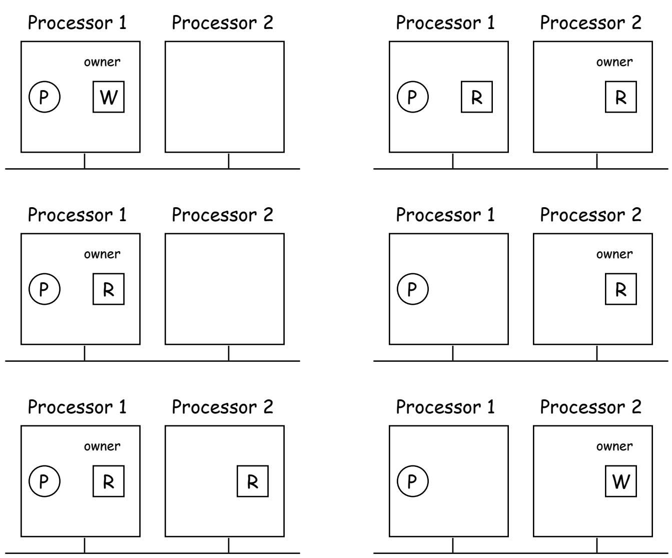
- **Update Options:** Write-Update *vs.* Write-Invalidate

- Write-Update:
  - Writes made locally are multicast to all copies of the data item.
  - Multiple writers can share same data item.
  - Consistency depends on multicast protocol.
    - E.g. Sequential consistency achieved with totally ordered multicast.
  - Reads are cheap
- Write-Invalidate:
  - Distinguish between read-only (multiple copies possible) and writable (only one copy possible).
  - When process attempts to write to remote or replicated item, it first sends a multicast message to invalidate copies; If necessary, get copy of item.
  - Ensures sequential consistency.

A Protocol for Sequential Consistency (reads)



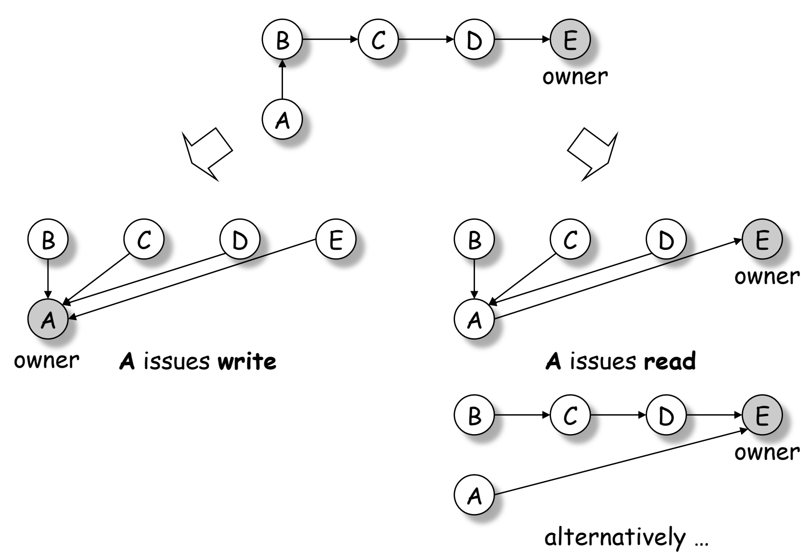
A Protocol for Sequential Consistency (write)



Design Issues (cont)

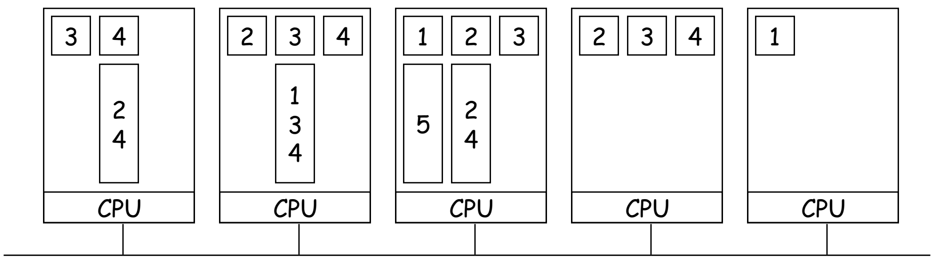
- **Finding the Owner**
  - broadcast request for owner
    - combine request with requested operation
    - problem: broadcast effects all participants (interrupts all processors), uses network bandwidth
  - page manager
    - possible hot spot
    - multiple page manager, hash on page address
  - probable owner
    - each process keeps track of probable owner
    - Update probable owner whenever
      - Process transfers ownership of a page
      - Process handles invalidation request for a page
      - Process receives read access for a page from another process
      - Process receives request for page it does not own (forwards request to probable owner and resets probable owner to requester)
    - periodically refresh information about current owners

Probable Owner Chains



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 in 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

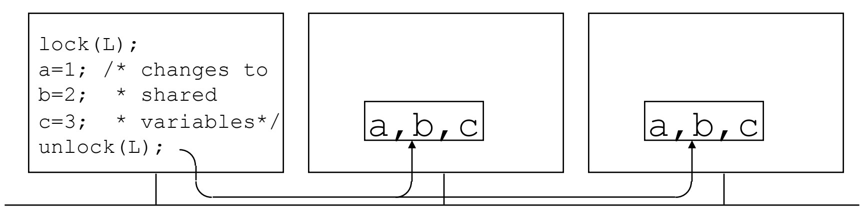
- Release consistency
- Multiple update protocols
- Directories for data location

Release Consistency in Munin/Treadmarks

- 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:



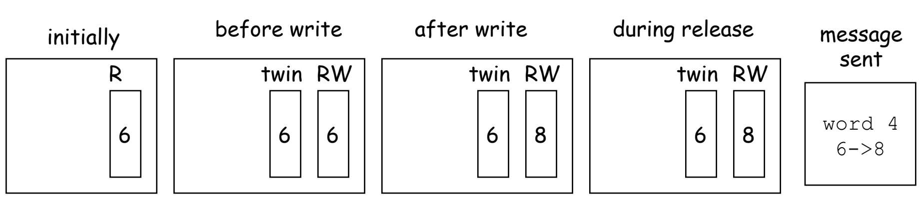
- **Eager** vs. **Lazy** Release Consistency

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.
  - others…

Twin Pages in Munin/Treadmarks

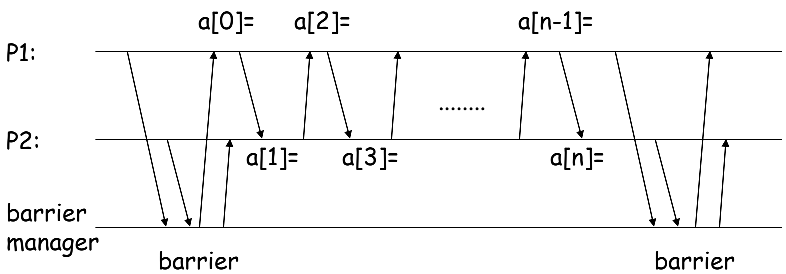
- Initially, write-shared page is marked as read-only.
- When **write** occurs, **twin copy** of page is made, and **original page** becomes read/write
- Release:
  1. word-by-word comparison of dirty pages with their twins
  2. send the **differences** to all processes needing them
  3. reset page to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



Effect of Using Twin Pages (no twins)

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);
```



Effect of Twin Pages (with twins)

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);
```