9. Linking and Sharing

9.1 Single-Copy Sharing
  – Why Share
  – Requirements for Sharing
  – Linking and Sharing

9.2 Sharing in Systems without Virtual Memory

9.3 Sharing in Paging Systems
  – Sharing of Data
  – Sharing of Code

9.3 Sharing in Segmented Systems

9.4 Principles of Distributed Shared Memory (DSM)
  – The User's View of DSM

9.5 Implementations of DSM
  – Implementing Unstructured DSM
  – Implementing Structured DSM

Single-Copy Sharing

- Focus: sharing a single copy of code or data in memory
- Why share?
  - Processes need to access common data
    - Producer/consumer, task pools, file directories
  - Better utilization of memory
    - code, system tables, data bases

Requirements for Sharing

- Requirement for sharing
  - How to express what is shared
    - *A priori* agreement (e.g., system components)
    - Language construct (e.g., UNIX's shmget/shmat)
  - Shared code must be *reentrant* (also known as *read-only* or *pure*)
    - Does not modify itself (read-only segments)
    - Data (stack, heap) in separate private areas for each process

Linking and Sharing

- Linking resolves external references
- Sharing links the *same copy* of a module into *two or more* address spaces
- Static linking/sharing:
  - Resolve references before execution starts
- Dynamic linking/sharing:
  - While executing

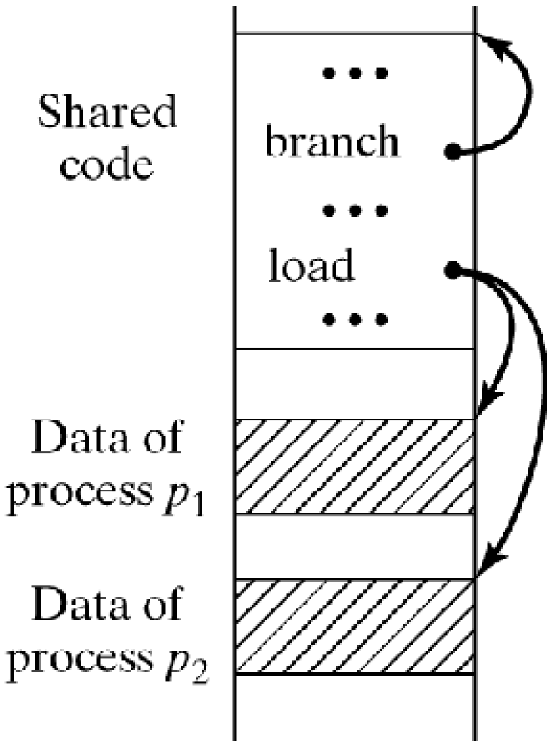

Figure 9-1

Sharing without Virtual Memory

- With one or no Relocation Register (RR)
  - All memory of a process is contiguous
  - Sharing user programs:
    - Possible only with 2 user programs by partial overlap
    - Too restrictive and difficult; generally not used
  - Sharing system components:
    - Components are assigned specific, agreed-upon starting positions
    - Linker resolves references to those locations
    - Can also use a block of transfer addresses, but this involves additional memory references.

Sharing without Virtual Memory

- With multiple RRs
  - CBR = Code Base Register
    Point to shared copy of code
  - SBR = Stack Base Register
    Point to private copy of stack
  - DBR = Data Base Register
    Point to private copy of data

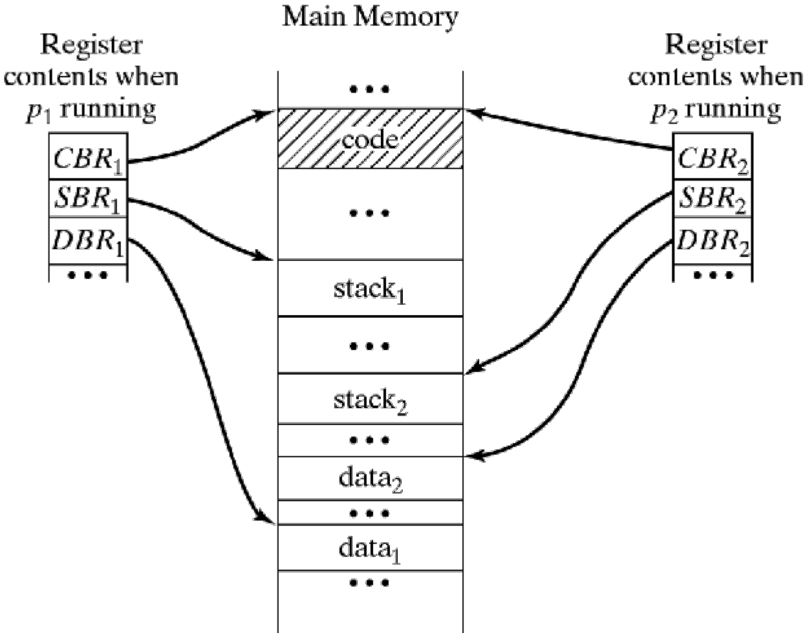

Figure 9-2

Sharing in Paging Systems

- Data pages
- Code pages
- Generally, want to avoid requiring shared page to have the same page number in all processes that share it
  - Code, data could be shared by many processes
  - Could easily lead to conflicts

Sharing in Paging Systems

- Sharing of data pages:
  - Page table entries of different processes point to the same page
  - If shared pages contain only data and no addresses, linker can
    - Assign arbitrary page numbers to the shared pages
    - Adjust page tables to point to appropriate page frames
  - So the shared page can have a different page number in different processes

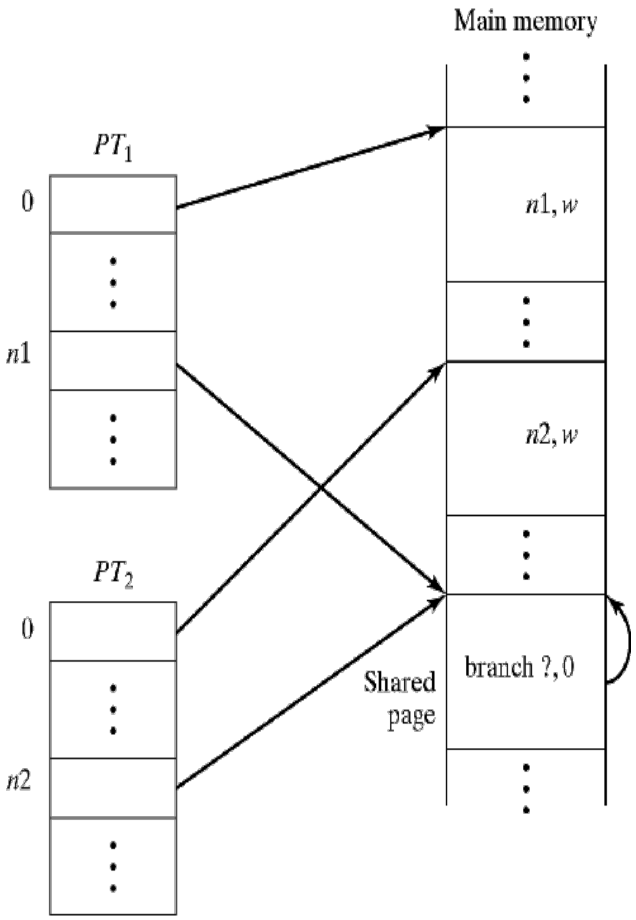

Figure 9-3

Sharing in Paging Systems

- Sharing of code pages
- Key issues:
  - Self-references: references to the shared code from within the shared code
  - Linking the shared code into multiple address spaces

Sharing of Code Pages in Paging Systems

- Self references:
  - avoid page numbers in shared code by compiling branch addresses relative to CBR
  - This works provided the shared code is *self-contained* (does not contain any external references)
- Linking shared pages into multiple address spaces:
  - Issues:
    - Want to defer loading of code until we actually use it
    - When process first accesses the code, it may have already been loaded by another process
  - Done through *dynamic linking* using a *transfer vector*

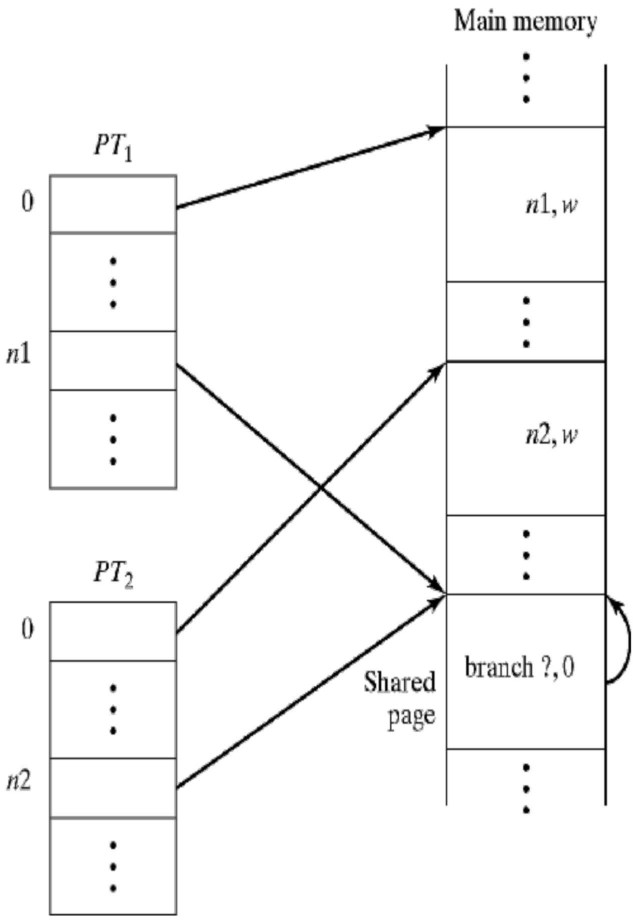

Figure 9-3

Dynamic Linking via Transfer Vector

- Each Transfer Vector entry corresponds to a reference to shared code
- Each entry initially contains a piece of code called a *stub*
- Stub code does the following:
  - Checks whether referenced shared code is loaded.
  - If the shared code is not already loaded, the stub loads the code
  - Stub code replaces itself by a direct reference to the shared code

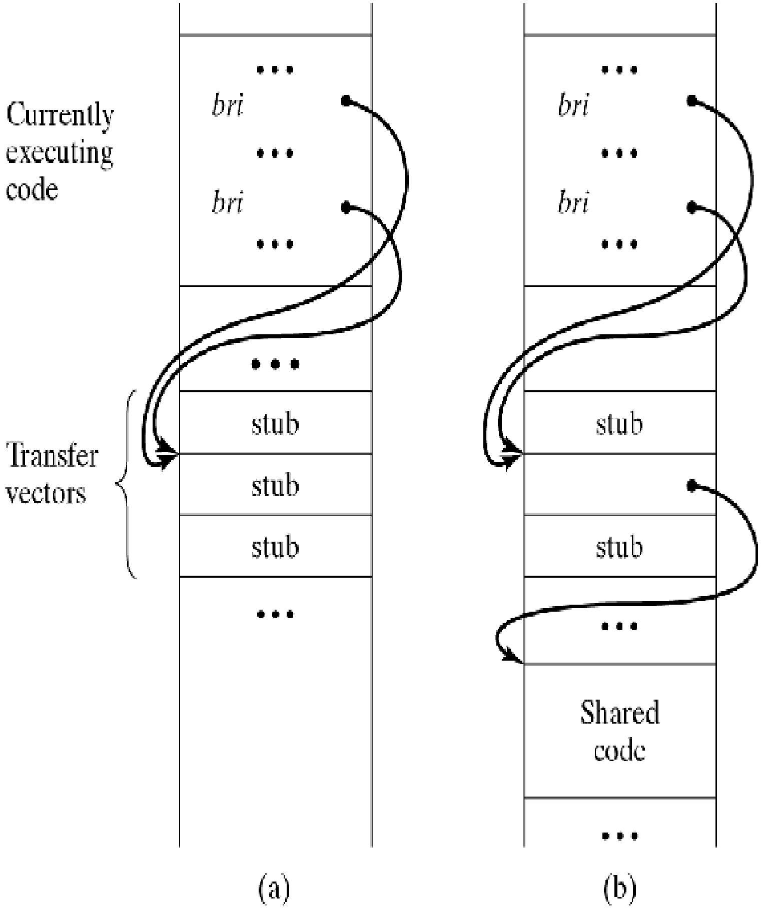

Figure 9-4

Sharing in Segmented Systems

- Much the same as with Paged Systems
- Simpler and more elegant because segments represent logical program entities
- ST entries of different processes point to the same segment in physical memory (PM)
- Data pages, containing only data and no addresses: same as with paged systems
- Code pages:
  - Assign same segment numbers in all STs, or
  - Use base registers:
    - Function call loads CBR
    - Self-references have the form ***w*(CBR*)***
    - This works if shared segments are *self-contained* (i.e., it they do not contain any references to other segments).
    - Full generality can be achieved using *private linkage sections*, introduced in Multics (1968).

Unrestricted Dynamic Linking/Sharing

- Basic Principles (see Figure 9-5 on next page):
  - Self-references resolved using CBR
  - External references are indirect via a private linkage section
  - External reference is (S,W), where S and W are symbolic names
  - At runtime, on first use:
    - Symbolic address (S,W) is resolved to (s,w), using trap mechanism)
    - (s,w) is entered in linkage section of process
    - Code is unchanged
  - Subsequent references use (s,w) without involving OS
  - Forces additional memory access for every external reference

Dynamic Linking/Sharing

Before and After External Reference is Executed

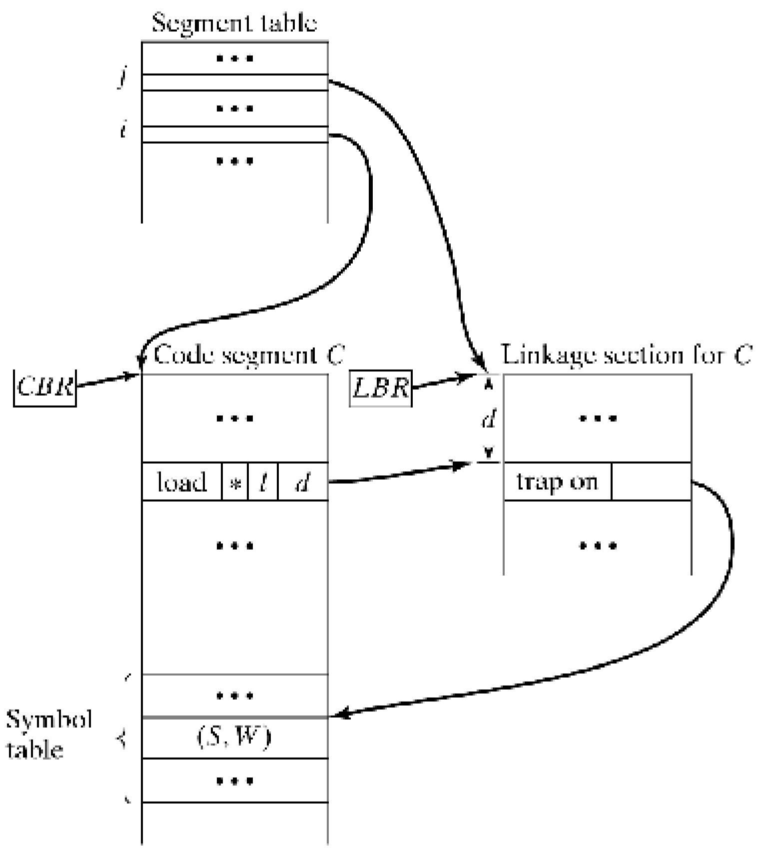

Figure 9-5a: Before

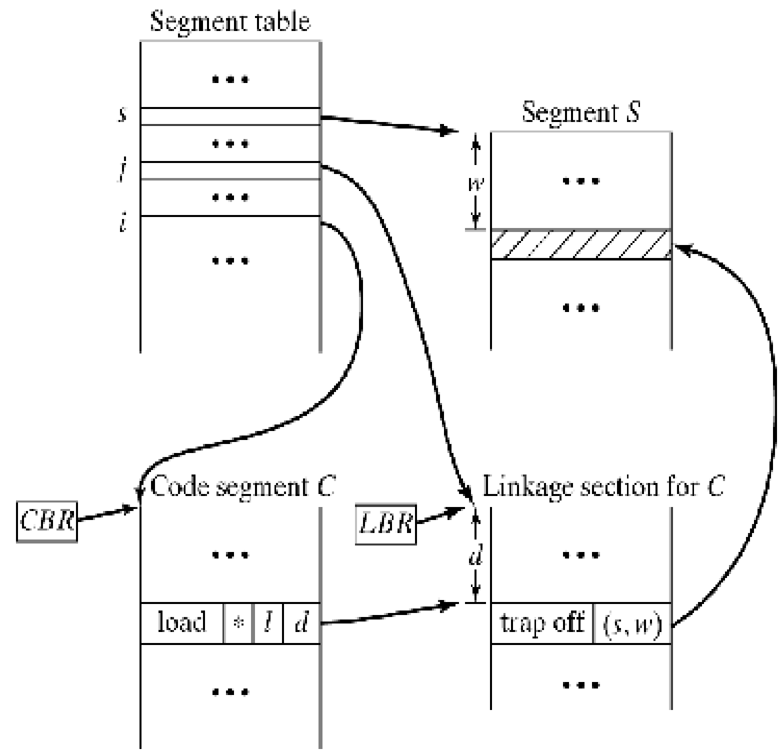

Figure 9-5b: After

Distributed Shared Memory

- Goal: Create illusion of single shared memory in a distributed system
- The (ugly) reality is that physical memory is distributed.
- References to remote memory trigger hidden transfers from remote memory to local memory
  - Impractical/Impossible to do this one reference at a time.
- How to implement transfers efficiently?
  - Optimize the implementation.
    Most important with Unstructured DSM.
  - Restrict the user. (Exploit what the user knows.)
    Basic to Structured DSM.

Unstructured DSM

Simulate single, fully shared, unstructured memory.
(Unlike paging, a CPU has no private space.)

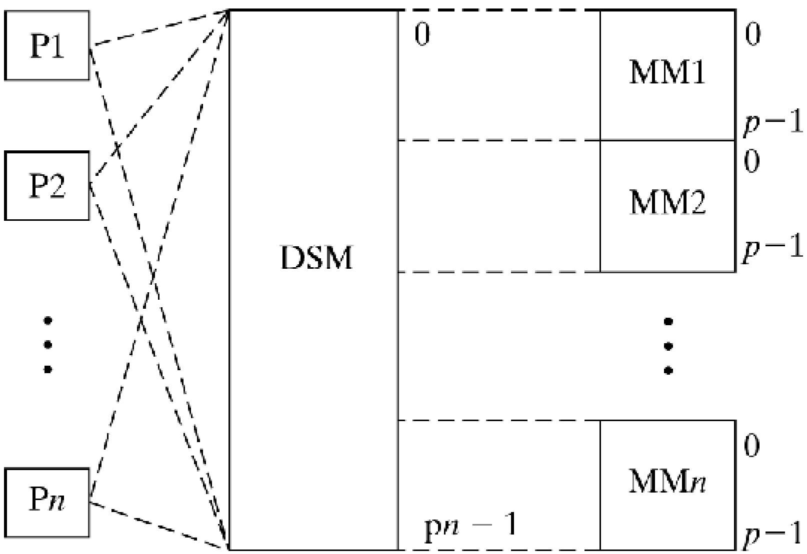

Figure 9-6

- Advantage: Fully transparent to user
- Disadvantage: Efficiency. Every instruction fetch or operand read/write could be to remote memory

Structured DSM

- Each CPU has both private and shared space.
- Add restrictions on use of shared variables:
  - Access only within (explicitly declared) Critical Sections
  - Modifications only need to be propagated at beginning/end of critical sections.
- Variant: Use objects instead of shared variables: *object-based DSM*

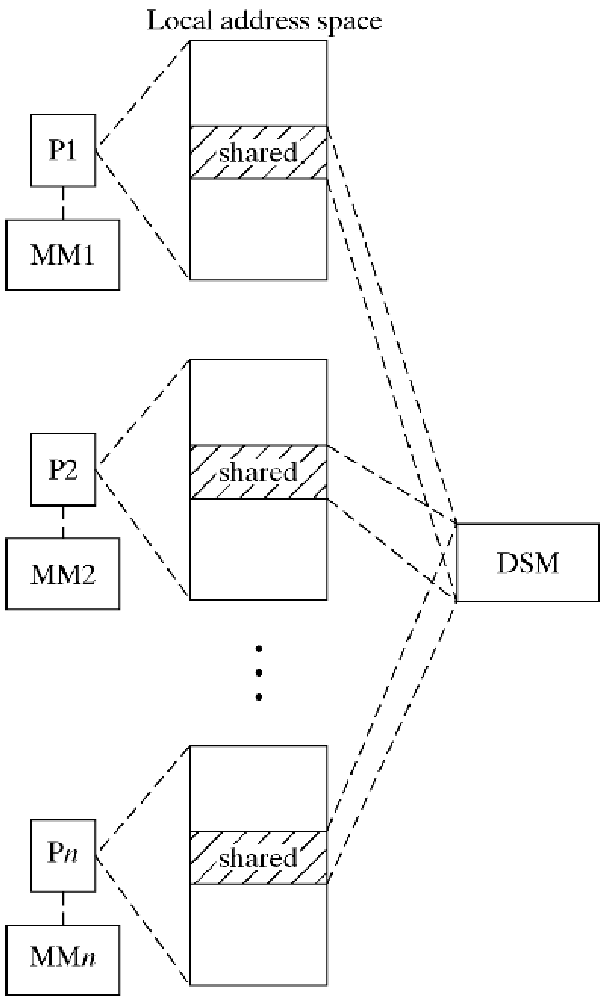

Figure 9-7

Implementing Unstructured DSM

- Key Issues:
  - Granularity of Transfers
  - Replication of Data
  - Memory Consistency: Strict *vs* Sequential
  - Tracking Data: Where is it stored now?

Implementing Unstructured DSM

- Granularity of Transfers
  - Transfer too little:
    - Time wasted in latency
  - Transfer too much:
    - Time wasted in transfer
    - *False sharing:*
      - Two unrelated variables, each accessed by a different process, are on the same page/set of pages being transferred between physical memories
      - Can result in pages being transferred back and forth, similar to thrashing

Implementing Unstructured DSM

- Replication of Data: Move or Copy?
  - Copying saves time on later references.
  - Copying causes (cache or real) consistency confusion.
    - Reads work fine.
    - Writes require others to update or invalidate.

Operation	Page Location	Page Status	Actions Taken Before Local Read/Write
read	local	read-only	
write	local	read-only	invalidate remote copies; upgrade local copy to writable
read	remote	read-only	make local read-only copy
write	remote	read-only	invalidate remote copies; make local writable copy
read	local	writable	
write	local	writable	
read	remote	writable	downgrade page to read-only; make local read-only copy
write	remote	writable	transfer remote writable copy to local memory

Figure 9-9

Implementing Unstructured DSM

- *Strict Consistency:* Reading a variable x returns the value written to x by the most recently executed write operation.
- *Sequential Consistency:* Sequence of values of x read by different processes corresponds to some sequential interleaved execution of those processes.

```
initial: x = 0
(a) p1:  {                        x = 1;    a1 = x;   x = 2;   b1 = x;}
    p2:  { a2 = x;      b2 = x;}

(b) p1:  {              x = 1;    a1 = x;   x = 2;    b1 = x;}
    p2:  { a2 = x;                b2 = x;}

(c) p1:  { x = 1;       a1 = x;   x = 2;    b1 = x;}
    p2:  {              a2 = x;                       b2 = x;}
```

Figure 9-11

  - Reads of x in p1 will always produce (1,2)
  - Reads of x in p2 can produce (0,0), (0,1), (0,2), (1,1), (1,2), or (2,2)

Implementing Unstructured DSM

- Tracking Data: Where is it stored now?
- Approaches:
  - Have owner track it by maintaining *copy set* (list). Only owner is allowed to write.
  - Ownership can change when a write request is received. Now we need to find the owner. ☺
    - Use broadcast.
    - *Central Manager* (→ Bottleneck). *Replicated managers* share responsibilities.
    - *Probable owner* gets tracked down. Retrace data's migration. Update links traversed to show current owner.
- Bottom line on Unstructured DSM:
  - Isn't there a better way?

Implementing Structured DSM

- Memory Consistency
  - Unstructured DSM assume that all shared variables are consistent at all times. This is a major reason why the performance is so poor.
  - Structured DSM introduces new, weaker models of memory consistency
    - Weak consistency
    - Release consistency
    - Entry consistency

Implementing Structured DSM

- Weak Memory Consistency
  - Introduce *synchronization variable* S
  - Processes access S when they are ready to adjust/reconcile their shared variables.
  - The DSM is only guaranteed to be in a consistent state immediately following access to a synchronization variable

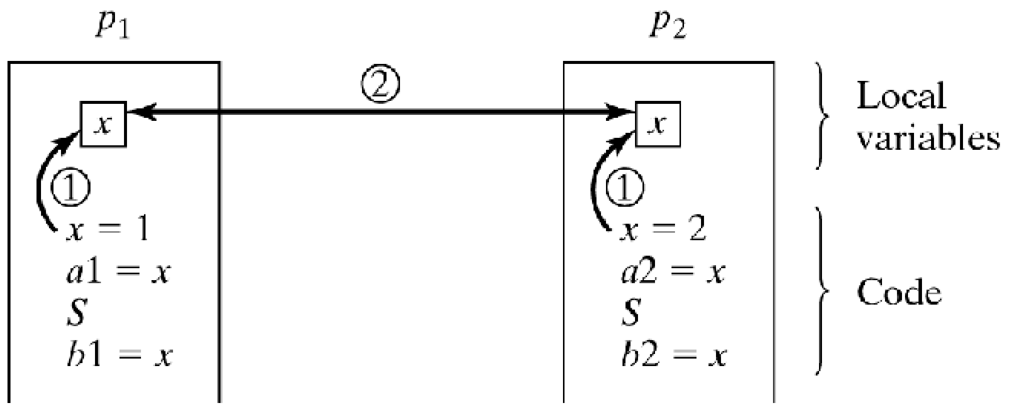

Figure 9-12

Implementing Structured DSM

- Release Memory Consistency
  - Export modified variables upon leaving CS

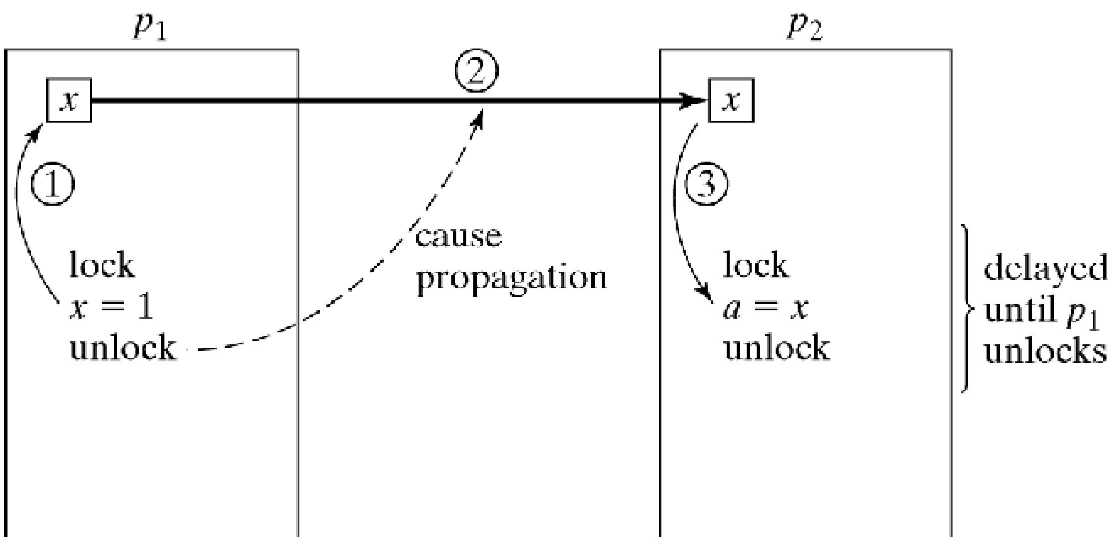

Figure 9-13

  - This is a waste if p2 never looks at x.

Implementing Structured DSM

- Entry Memory Consistency
  - Associate each shared variable with a lock variable
  - Before entering CS, import only those variables associated with the current lock

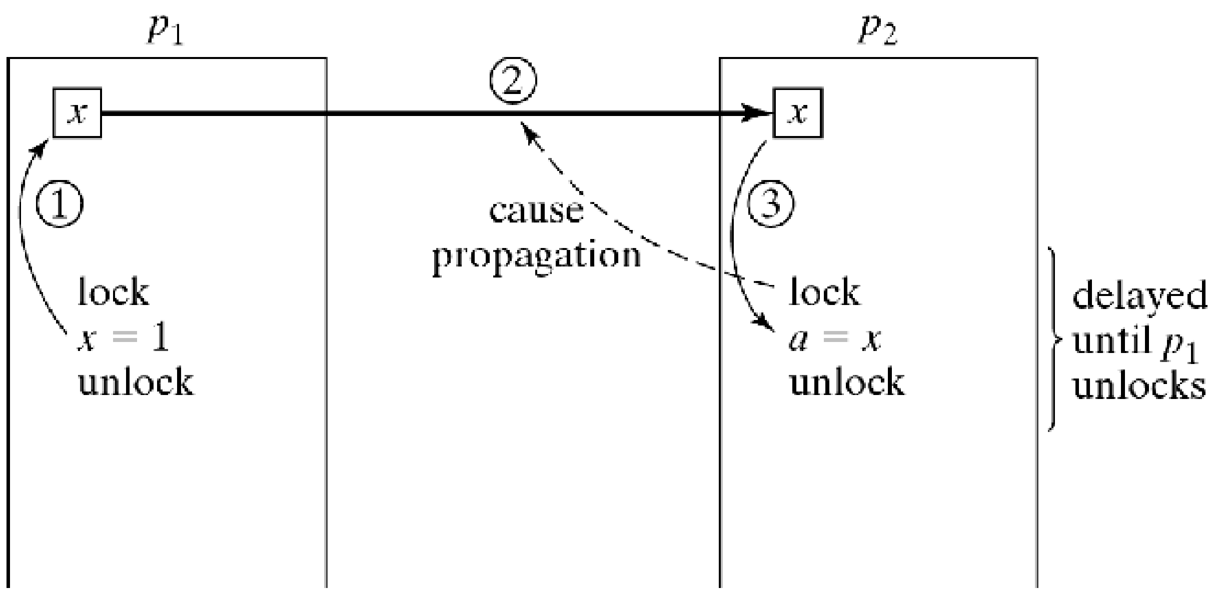

Figure 9-14

- There is also a (confusingly named?) *lazy release* consistency model which imports all shared variables before entering CS

Object-Based DSM

- An object's functions/methods are part of it.
- Can use remote method invocation
  (like remote procedure calls, covered earlier)
  instead of copying or moving an object into
  local memory.
- Can also move or copy an object to improve performance.
- When objects are replicated, consistency issues again arise
  (as in unstructured DSM)
- On write, we could
  - Invalidate all other copies (as in unstructured DSM)
  - Remotely invoke, on all copies, a method that does the same write

Memory Models on Multiprocessors

- Processors share memory
- Each processor may have its own cache
- Memory models provide rules for deciding
  - When processor X sees writes to memory by other processors
  - When writes by processor X are visible to other processors
- These questions are similar to some of the issues that arise in distributed shared memory
-

Java Memory Model

Similar issues arise in multithreaded code in Java

- Each thread may have its own copy of shared variables
- Threads may read from and write to their own copy of shared variables.
- The Java Memory Model specifies
  - When thread X must see writes to memory by other processors
  - When writes by thread X must become visible to other processors
- The issues in Java are different from those in other languages such as C/C++:
  - Threads are an integral part of the Java language.
  - Java compilers can rearrange thread code as part of optimization
  - To achieve correctness, certain conditions must be guaranteed.

Java Memory Model (continued)

- Full details in JSR 133 (2004).
- A *happened-before* relation is defined on memory references, locks, unlocks, and other thread operation.
- If one action happened-before the other according to this definition, then the Java Virtual Machine guarantees that the results of the first action are visible to the second action
- **Example:**
  - If x=1 happened-before y=x and no other assignment to x intervenes, then y must be set to 1.
  - But if it is not true that x=1 happened-before y=x, then y will not necessarily be set to 1.
- Note that this is a separate issue from mutual exclusion, although the two are related.

Java Memory Model (continued)

- Some rules defining the happened before relation (not a complete list):
  - An action in a thread happened-before an action in that thread that comes later in the thread's sequential order.
  - An unlock on an object happened-before every subsequent lock on **that same object**.
  - A write to a volatile field happened-before every subsequent read of **that same volatile field**.
  - A call to start() on a thread happened-before any actions within the thread.
  - All actions within a thread happened-before any other thread returns from a join() on that thread.
  - A write by a thread to a blocking queue happened-before any subsequent read from that blocking queue.
- There are other rules. The compiler is free to reorder operations as long as the happened-before operation is respected.

History

- Originally developed by Steve Franklin
- Modified by Michael Dillencourt, Summer, 2007
- Modified by Michael Dillencourt, Spring, 2009
- Modified by Michael Dillencourt, Winter 2011 (added material on Java memory model)