Lecture 18: Multiprocessors 2: Snooping v. Directory Coherency, Memory Consistency Models

Professor David A. Patterson
Computer Science 252
Spring 1998
Review: Parallel Framework

- Layers:

  - Programming Model:
    » Multiprogramming: lots of jobs, no communication
    » Shared address space: communicate via memory
    » Message passing: send and receive messages
    » Data Parallel: several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)

  - Communication Abstraction:
    » Shared address space: e.g., load, store, atomic swap
    » Message passing: e.g., send, receive library calls
    » Debate over this topic (ease of programming, scaling)
      => many hardware designs 1:1 programming model
Review : Small-Scale MP Designs

- Memory: centralized with uniform access time ("uma") and bus interconnect
- Examples: Sun Enterprise 5000, SGI Challenge, Intel SystemPro
Distributed Directory MPs
Revised Snoopy-Cache State Machine

- State machine for **CPU and bus** requests for each memory block
- Invalid state if in memory

- Invalid state if in memory
- Remote Write or Miss due to address conflict
- Write back block

- CPU Read hit

- CPU Write
- Place Write Miss on bus
- Remote Read
- Write back block

- CPU Read hit

- Shared (read/only)

- Invalid
Snoop Cache Extensions

Extensions:
- **Fourth State: Ownership**
  - Shared-> Modified, need invalidate only (upgrade request), don’t read memory
  - Berkeley Protocol
- **Clean exclusive state** (no miss for private data on write)
  - MESI Protocol
  - Cache supplies data when shared state (no memory access)
  - Illinois Protocol

**State Transitions:**
- **Invalid**
  - Remote Write or Miss due to address conflict
  - CPU Read

- **Shared** (read/only)
  - CPU Read hit
  - Remote Write or Miss due to address conflict
  - CPU Write

- **Modified** (read/write)
  - CPU Read hit
  - CPU Write
  - Place Write Miss on bus
  - Remote Read

- **Exclusive** (read/only)
  - CPU Write
  - Place Write Miss on Bus?
  - CPU Read hit
### Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>step</strong></td>
<td><strong>P1</strong></td>
<td><strong>P2</strong></td>
<td><strong>Bus</strong></td>
</tr>
<tr>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>State</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but $A1 \neq A2$
Example: Step 1

<table>
<thead>
<tr>
<th>Step</th>
<th>P1: Write 10 to A1</th>
<th>P1: Read A1</th>
<th>P2: Read A1</th>
<th>P2: Write 20 to A1</th>
<th>P2: Write 40 to A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Excl.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>WrMs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proc. Addr</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2. Active arrow =

Diagram:

- Invalid
- Shared
- Exclusive
- Remote Write or Miss
- Write miss on bus
- Remote Read
- Place Write Miss on Bus
- CPU Read hit
- CPU Write
- CPU read hit
- CPU write hit
Example: Step 2

<table>
<thead>
<tr>
<th>Step</th>
<th>P1 State</th>
<th>P1 Addr</th>
<th>P1 Value</th>
<th>P2 State</th>
<th>P2 Addr</th>
<th>P2 Value</th>
<th>Bus Action</th>
<th>Proc. Addr</th>
<th>Value</th>
<th>Memory Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2
### Example: Step 3

<table>
<thead>
<tr>
<th>Step</th>
<th>P1</th>
<th>P2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>State</td>
<td>Addr</td>
<td>Value</td>
<td>State</td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2.
### Example: Step 4

<table>
<thead>
<tr>
<th>Step</th>
<th>P1 State</th>
<th>P1 Addr</th>
<th>P1 Value</th>
<th>P2 State</th>
<th>P2 Addr</th>
<th>P2 Value</th>
<th>Action</th>
<th>Proc. Addr</th>
<th>Value</th>
<th>Memory Addr</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>RdMs</td>
<td>P2</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>WrBk</td>
<td>P1</td>
<td>A1</td>
<td>10</td>
<td>A1 10</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Inv.</td>
<td>Excl.</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td>P2</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2
### Example: Step 5

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>WrBk</td>
<td>P1</td>
<td>A1 10</td>
<td>A1 10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>RdDa</td>
<td>P2</td>
<td>A1 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Inv.</td>
<td>A1</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td>P2</td>
<td>A1 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Excl.</td>
<td>A2</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td>P2</td>
<td>A2 10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2
CS 252 Administrivia

Wed 15-Apr Project Reviews: 8-12:30, 3-5:10 (no lecture)

Fri 17-Apr Searching the Computer Science Literature: Techniques & Tips by Camille Wanat, Eng. Library

Sun 19-Apr Quiz Review 2-3PM?, 306? Soda

Wed 22-Apr Quiz #2 5:30-8:30 (no lecture); Pizza after

Fri 24-Apr “How to have a Bad Academic Career”
Also read Technology and Courage, by Ivan Sutherland, Sun Microsystems

Wed 29-Apr (no lecture)

Thu 30-Apr 1-7PM; Final Oral Presentation (30 Min)

Fri 1-May 1-5PM; Final Oral Presentation (no lecture)

Wed 6-May 1:30-3:30; Public Poster Session 6th floor

Fri 8-May Last lecture; Goodbye to Architecture

Mon 11-May URLs of Projects due
Snooping Coherency Implementation Complications

• Write Races:
  – Cannot update cache until bus is obtained
    » Otherwise, another processor may get bus first, and then write the same cache block!
  – Two step process:
    » Arbitrate for bus
    » Place miss on bus and complete operation
  – If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  – Split transaction bus:
    » Bus transaction is not atomic:
      can have multiple outstanding transactions for a block
    » Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    » Must track and prevent multiple misses for one block

• Must support interventions and invalidations
Implementing Snooping Caches

• Multiple processors must be on bus, access to both addresses and data
• Add a few new commands to perform coherency, in addition to read and write
• Processors continuously snoop on address bus
  – If address matches tag, either invalidate or update
• Since every bus transaction checks cache tags, could interfere with CPU just to check:
  – solution 1: duplicate set of tags for L1 caches to allow checks in parallel with CPU
  – solution 2: L2 cache already duplicate and underutilized, provided L2 obeys inclusion with L1 cache
    » block size, associativity of L2 affects L1
Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)
Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- 1 Cache Coherency solution: non-cached pages
- Alternative: directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is $f(\text{memory size})$ vs. $f(\text{cache size})$
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which Procs have copies of their blocks
Directory Protocol

- Similar to Snoopy Protocol: Three states
  - **Shared**: ≥ 1 processors have data, memory up-to-date
  - **Uncached**: (no processor has it; not valid in any cache)
  - **Exclusive**: 1 processor (owner) has data; memory out-of-date

- In addition to cache state, must track which **processors** have data when in the shared state (usually bit vector, 1 if processor has copy)

- Keep it simple(r):
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent
Directory Protocol

• No bus and don’t want to broadcast:
  – interconnect no longer single arbitration point
  – all messages have explicit responses

• Terms: typically 3 processors involved
  – Local node where a request originates
  – Home node where the memory location of an address resides
  – Remote node has a copy of a cache block, whether exclusive or shared

• Example messages on next slide:
  \( P = \) processor number, \( A = \) address
# Directory Protocol Messages

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Msg Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Processor P reads data at address A;</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>make P a read sharer and arrange to send data back</em></td>
</tr>
<tr>
<td>Write miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Processor P writes data at address A;</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>make P the exclusive owner and arrange to send data back</em></td>
</tr>
<tr>
<td>Invalidate</td>
<td>Home directory</td>
<td>Remote caches</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Invalidate a shared copy at address A.</em></td>
</tr>
<tr>
<td>Fetch</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Fetch the block at address A and send it to its home directory</em></td>
</tr>
<tr>
<td>Fetch/Invalidate</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Fetch the block at address A and send it to its home directory; invalidate the block in the cache</em></td>
</tr>
<tr>
<td>Data value reply</td>
<td>Home directory</td>
<td>Local cache</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Return a data value from the home memory (read miss response)</em></td>
</tr>
<tr>
<td>Data write-back</td>
<td>Remote cache</td>
<td>Home directory</td>
<td>A, Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Write-back a data value for address A (invalidate response)</em></td>
</tr>
</tbody>
</table>
State Transition Diagram for an Individual Cache Block in a Directory Based System

- States identical to snoopy case; transactions very similar.
- Transitions caused by read misses, write misses, invalidates, data fetch requests.
- Generates read miss & write miss msg to home directory.
- Write misses that were broadcast on the bus for snooping => explicit invalidate & data fetch requests.
- Note: on a write, a cache block is bigger, so need to read the full cache block.
CPU -Cache State Machine

- State machine for CPU requests for each memory block
- Invalid state if in memory

CPU read hit
CPU write hit

Invalid state:
- Send Data Write Back message to home directory

CPU Read hit

CPU Read:
- Send Read Miss message

CPU Write:
- Send Write Miss message to home directory

Fetch:
- Send Data Write Back message to home directory

Fetch/Invalidate or Miss due to address conflict:
- Send Data Write Back message to home directory

Invalidate or Miss due to address conflict:
- Send Read Miss message

Shared (read/only)

Exclusive (read/write)
State Transition Diagram for the Directory

• Same states & structure as the transition diagram for an individual cache
• 2 actions: update of directory state & send msgs to statisfy requests
• Tracks all copies of memory block.
• Also indicates an action that updates the sharing set, Sharers, as well as sending a message.
**Directory State Machine**

- **State machine for** *Directory* **requests for each memory block**
- **Uncached state if in memory**

**Data Write Back:**
- Sharers = {}
- *(Write back block)*

**Write Miss:**
- Sharers = {P}; send Data Value Reply msg
- Replication not required, go to Read

**Read:**
- Sharers += {P}; send Data Value Reply msg to remote cache
- *(Write back block)*

**Write Miss:**
- Sharers += {P}; send Fetch/Invalidate; send Data Value Reply msg to remote cache

**Read miss:**
- Sharers += {P}; send Data Value Reply msg

**Write Miss:**
- Sharers += {P}; send Fetch; send Data Value Reply msg to remote cache
- *(Write back block)*

**Read miss:**
- Sharers += {P}; send Data Value Reply msg

**Write Miss:**
- Sharers += {P}; send Invalidate to Sharers; then Sharers = {P}; send Data Value Reply msg

**Read:**
- Sharers += {P}; send Data Value Reply msg

**Write Miss:**
- Sharers += {P}; send Fetch/Invalidate; send Data Value Reply msg

**Read:**
- Sharers += {P}; send Data Value Reply msg

**Write Miss:**
- Sharers += {P}; send Fetch; send Data Value Reply msg to remote cache
- *(Write back block)*
Example Directory Protocol

• Message sent to directory causes two actions:
  – Update the directory
  – More messages to satisfy request

• Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  – **Read miss**: requesting processor sent data from memory & requestor made only sharing node; state of block made Shared.
  – **Write miss**: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

• Block is Shared => the memory value is up-to-date:
  – **Read miss**: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  – **Write miss**: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.
Example Directory Protocol

- **Block is Exclusive**: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
  - **Read miss**: owner processor sent data fetch message, causing state of block in owner’s cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor. Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
  - **Data write-back**: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
  - **Write miss**: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.
Example

A1 and A2 map to the same cache block

<table>
<thead>
<tr>
<th>step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>P1: Read A1</td>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example

#### Processor 1 Processor 2 Interconnect Directory Memory

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td>A1</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl. A1 10</td>
<td></td>
<td>DaRp</td>
<td>P1</td>
<td>A1 0</td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
### Example

A1 and A2 map to the same cache block
Example

A1 and A2 map to the same cache block

<table>
<thead>
<tr>
<th>step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl. A1 10</td>
<td>WrMs P1 A1</td>
<td>A1 Ex {P1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl. A1 10</td>
<td>DaRp P1 A1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar. A1 10</td>
<td>RdMs P2 A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Shar. A1 10</td>
<td>DaRp P2 A1 10 A1 Shar. {P1,P2} 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write Back
### Example

A1 and A2 map to the same cache block.

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl. A1 10</td>
<td>DaRp P1 A1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl. A1 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar. A1 10</td>
<td>Ftch P1 A1 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl. A1 20</td>
<td>WrMs P2 A1 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example

A1 and A2 map to the same cache block

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl. A1 10</td>
<td>DaRp</td>
<td>P1 A1</td>
<td>A1</td>
<td>Ex {P1}</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl. A1 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar. A1 10</td>
<td>Ftch</td>
<td>P1 A1 10</td>
<td>A1</td>
<td>Shar. {P1,P2} 10</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl. A1 20</td>
<td>WrMs</td>
<td>P2 A1</td>
<td>A1</td>
<td>Excl. {P2} 10</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Excl. A2 40</td>
<td>DaRp</td>
<td>P2 A2 0</td>
<td>A2</td>
<td>Excl. {P2} 20</td>
</tr>
</tbody>
</table>
Implementing a Directory

• We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network (see Appendix E)

• Optimizations:
  – read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor
Synchronization

• Why Synchronize? Need to know when it is safe for different processes to use shared data

• Issues for Synchronization:
  – Uninterruptable instruction to fetch and update memory (atomic operation);
  – User level synchronization operation using this primitive;
  – For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization
Uninterruptable Instruction to Fetch and Update Memory

- **Atomic exchange**: interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    - 0 if you succeeded in setting the lock (you were first)
    - 1 if other processor had already claimed access
  - Key is that exchange operation is indivisible

- **Test-and-set**: tests a value and sets it if the value passes the test

- **Fetch-and-increment**: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free
Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- **Load linked** (or load locked) + **store conditional**
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise

- Example doing atomic swap with LL & SC:
  ```
  try: mov R3,R4 ; mov exchange value
  ll R2,0(R1) ; load linked
  sc R3,0(R1) ; store conditional
  beqz R3,try ; branch store fails (R3 = 0)
  mov R4,R2 ; put load value in R4
  ```

- Example doing fetch & increment with LL & SC:
  ```
  try: ll R2,0(R1) ; load linked
  addi R2,R2,#1 ; increment (OK if reg–reg)
  sc R2,0(R1) ; store conditional
  beqz R2,try ; branch store fails (R2 = 0)
  ```
User Level Synchronization—Operation Using this Primitive

- **Spin locks**: processor continuously tries to acquire, spinning around a loop trying to get the lock
  
  ```
  li   R2,#1
  lockit:  exch  R2,0(R1)   ;atomic exchange
          bnez  R2,lockit   ;already locked?
  ```

- **What about MP with cache coherency?**
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables

- **Problem**: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

- **Solution**: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):
  
  ```
  try:   li   R2,#1
  lockit:  lw  R3,0(R1)   ;load var
          bnez  R3,lockit   ;not free=>spin
          exch  R2,0(R1)   ;atomic exchange
          bnez  R2,try     ;already locked?
  ```
Another MP Issue: Memory Consistency Models

• What is consistency? When must a processor see the new value? e.g., seems that

P1: \( A = 0; \) \hspace{1cm} P2: \( B = 0; \)

\( \ldots \)

A = 1; \hspace{1cm} \ldots \hspace{1cm} B = 1;

L1: if \( B == 0 \) \ldots \hspace{1cm} L2: if \( A == 0 \) \ldots

• Impossible for both if statements L1 & L2 to be true?
  – What if write invalidate is delayed & processor continues?

• Memory consistency models: what are the rules for such cases?

• Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
  – SC: delay all memory accesses until all invalidates done
Memory Consistency Model

• Schemes faster execution to sequential consistency

• Not really an issue for most programs; they are synchronized
  – A program is synchronized if all access to shared data are ordered by synchronization operations
    write \( x \)
    
    ...  
    
    release \( s \) \{unlock\}
    
    ...  
    
    acquire \( s \) \{lock\}
    
    ...  
    
    read(x)

• Only those programs willing to be nondeterministic are not synchronized: “data race”: outcome \( f(\text{proc. speed}) \)

• Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses
Review

• Caches contain all information on state of cached memory blocks

• Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)

• Directory has extra data structure to keep track of state of all cache blocks

• Distributing directory => scalable shared address multiprocessor => Cache coherent, Non uniform memory access