The Cache-Coherence Problem
Lecture 12
(Chapter 6)
Outline

• Bus-based multiprocessors
• The cache-coherence problem
• Peterson’s algorithm
• Coherence vs. consistency
Shared vs. Distributed Memory

• **What is the difference** between …
  – SMP
  – NUMA
  – Cluster ?
Small to Large Multiprocessors

• **Small scale** (2–30 processors): shared memory
  – Often on-chip: shared memory (+ perhaps shared cache)
  – Most processors have MP support out of the box
  – Most of these systems are bus-based
  – Popular in commercial as well as HPC markets

• **Medium scale** (64–256): shared memory and clusters
  – Clusters are cheaper
  – Often, clusters of SMPs

• **Large scale** (> 256): few shared memory and many clusters
  – SGI Altix 3300: 512-processor shared memory (NUMA)
  – Large variety on custom/off-the-shelf components such as interconnection networks.
    • Beowulf clusters: fast Ethernet
    • Myrinet: fiber optics
    • IBM SP2: custom
Shared Memory vs. No Shared Memory

- Advantages of shared-memory machines (vs. distributed memory w/same total memory size)
  - Support shared-memory programming
  - Clusters can also support it via *software shared virtual memory*, but with much coarser granularity and higher overheads
  - Allow fine-grained sharing
    - You can’t do this with messages—there’s too much overhead to share small items
  - Single OS image
- Disadvantage of shared-memory machines
  - Cost of providing shared-memory abstraction
A Bus-Based Multiprocessor

<table>
<thead>
<tr>
<th>CPU</th>
<th>P-Pro module</th>
<th>P-Pro module</th>
<th>P-Pro module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt controller</td>
<td>256-KB L2 $</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bus interface</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P-Pro bus (64-bit data, 36-bit address, 66 MHz)

- PCI bridge
- PCI bus
- Memory controller
- 1-, 2-, or 4-way interleaved DRAM

PCI bus

PCI I/O cards
Outline

• Bus-based multiprocessors
• The cache-coherence problem
• Peterson’s algorithm
• Coherence vs. consistency
Will This Parallel Code Work Correctly?

```c
sum = 0;
begin parallel
for (i=1; i<=2; i++) {
    lock(id, myLock);
    sum = sum + a[i];
    unlock(id, myLock);
}
end parallel
print sum;
```


Two issues:
- Will it print `sum = 10`?
- How can it support locking correctly?
sum = 0;
begin parallel
for (i=1; i<=2; i++) {
    lock(id, myLock);
    sum = sum + a[i];
    unlock(id, myLock);
}
end parallel
print sum;

Suppose \( a[1] = 3 \) and \( a[2] = 7 \)

• Will it print \( \text{sum} = 10 \)?
Cache-Coherence Problem Illustration

Start state. All caches empty and main memory has \( \text{Sum} = 0 \).

Trace

<table>
<thead>
<tr>
<th>Process</th>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>Read Sum</td>
<td></td>
</tr>
<tr>
<td>( P_1 )</td>
<td>Write Sum</td>
<td>3</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>Read Sum</td>
<td></td>
</tr>
<tr>
<td>( P_2 )</td>
<td>Write Sum</td>
<td>7</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>Read Sum</td>
<td></td>
</tr>
</tbody>
</table>
P₁ reads *Sum* from memory.

Trace

- P₁ Read Sum
- P₂ Read Sum
- P₁ Write Sum = 3
- P₂ Write Sum = 7
- P₁ Read Sum
$P_2$ reads. Let’s assume this comes from memory too.

Trace

- $P_1$ Read $Sum$
- $P_2$ Read $Sum$
- $P_1$ Write $Sum = 3$
- $P_2$ Write $Sum = 7$
- $P_1$ Read $Sum$
Cache-Coherence Problem Illustration

$P_1$ writes. This write goes to the cache.
P2 writes.

**Trace**
- P1 Read Sum
- P2 Read Sum
- P1 Write Sum = 3
- **P2 Write Sum = 7**
- P1 Read Sum
Cache-Coherence Problem Illustration

P₁ reads.

Trace

<table>
<thead>
<tr>
<th>P₁</th>
<th>Read Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₂</td>
<td>Read Sum</td>
</tr>
<tr>
<td>P₁</td>
<td>Write Sum = 3</td>
</tr>
<tr>
<td>P₂</td>
<td>Write Sum = 7</td>
</tr>
</tbody>
</table>

Controller

Sum = 0

Main memory

Bus

P₁ | Cache
---|---
Sum = 3 D

P₂ | Cache
---|---
Sum = 7 D

P₃ | Cache
---|---
Cache-Coherence Problem

- Do P1 and P2 see the same sum?

- Does it matter if we use a WT cache?

- The code given at the start of the animation does not exhibit the same coherence problem shown in the animation. Explain. Is the result still incoherent?

- What if we do not have caches, or sum is uncacheable. Will it work?
Write-Through Cache Does Not Work

$p_1$ reads.

- $p_1$ reads.
- $p_2$ reads.
- $p_1$ writes $\text{Sum} = 3$
- $p_2$ writes $\text{Sum} = 7$
- $p_1$ reads $\text{Sum}$

<table>
<thead>
<tr>
<th>Trace</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>Read $\text{Sum}$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>Read $\text{Sum}$</td>
</tr>
<tr>
<td>$p_1$</td>
<td>Write $\text{Sum} = 3$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>Write $\text{Sum} = 7$</td>
</tr>
</tbody>
</table>

**Controller**
- $\text{Sum} = 7$

**Main memory**

**Cache**
- $p_1$ cache: $\text{Sum} = 3$
- $p_2$ cache: $\text{Sum} = 7$
- $p_3$ cache

**Bus**

$\text{Controller}$

$\text{Main memory}$
Here’s simple code to implement a lock:

```c
void lock (int process, int lvar) { // process is 0 or 1
    while (lvar == 1) {} ;
    lvar = 1;
}

void unlock (int process, int lvar) {
    lvar = 0;
}
```

Will this guarantee mutual exclusion?

Let’s look at an algorithm that will …
Outline

• Bus-based multiprocessors
• The cache-coherence problem
• Peterson’s algorithm
• Coherence vs. consistency
Peterson’s Algorithm

```c
int turn;
int interested[n]; // initialized to false

void lock (int process, int lvar) { // process is 0 or 1
    int other = 1 - process;
    interested[process] = TRUE;
    turn = other;
    while (turn == other && interested[other] == TRUE) {};
} // Post: turn != other or interested[other] == FALSE

void unlock (int process, int lvar) {
    interested[process] = FALSE;
}
```

- Acquisition of `lock()` occurs only if
  1. `interested[other] == FALSE`: either the other process has not competed for the lock, or it has just called `unlock()`, or
  2. `turn != other`: the other process is competing, has set the turn to our process, and will be blocked in the `while()` loop
// Proc 0
interested[0] = TRUE;
turn = 1;
while (turn==1 && interested[1]==TRUE)
{
    // since interested[1] starts out FALSE,
    // Proc 0 enters critical section
}

// Proc 1
interested[1] = TRUE;
turn = 0;
while (turn==0 && interested[0]==TRUE)
{
    // since turn==0 && interested[0]==TRUE
    // Proc 1 waits in the loop until Proc 0
    // releases the lock
}

// unlock
interested[0] = FALSE;

// now Proc 1 can exit the loop and
// acquire the lock
// Proc 0
interested[0] = TRUE;
turn = 1;

while (turn==1 && interested[1]==TRUE)
    {};
// since turn == 0,
// Proc 0 enters critical section

// unlock
interested[0] = FALSE;

// Proc 1
interested[1] = TRUE;

turn = 0;

while (turn==0 && interested[0]==TRUE)
    {};
// since turn==0 && interested[0]==TRUE
// Proc 1 waits in the loop until Proc 0
// releases the lock

// now Proc 1 can exit the loop and
// acquire the lock
When Does Peterson’s Alg. Work?

- Correctness depends on the global order of

```
A: interested[process] = TRUE;
B: turn = other;
```

- Thus, it will not work if—
  - The *compiler* reorders the operations
    - There’s no data dependence, so unless the compiler is notified, it may well reorder the operations
    - This prevents compiler from using aggressive optimizations used in serial programs
  - The *architecture* reorders the operations
    - Write buffers, memory controller
    - Network delay for statement A
    - If `turn` and `interested[]` are cacheable, A may result in cache miss, but B in cache hit
  - This is called the memory-consistency problem.
Race on a Non-Sequentially Consistent Machine

// Proc 0
interested[0] = TRUE;

turn = 1;
while (turn==1 && interested[1]==TRUE)
{
}

// Proc 1
interested[1] = TRUE;

turn = 0;
while (turn==0 && interested[0]==TRUE)
{
}
Race on a Non-Sequentially Consistent Machine

// Proc 0
interested[0] = TRUE;

turn = 1;
while (turn==1 && interested[1]==TRUE)
{
} // since interested[1] == FALSE,
// Proc 0 enters critical section

// Proc 1
turn = 0;
interested[1] = TRUE;
while (turn==0 && interested[0]==TRUE)
{
} // since turn==1,
// Proc 1 enters critical section

Can you explain what has gone wrong here?
<table>
<thead>
<tr>
<th>Cache coherence</th>
<th>Memory consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deals with the ordering of operations to a <em>single</em> memory location.</td>
<td>Deals with the ordering of operations to <em>different memory locations</em>.</td>
</tr>
</tbody>
</table>
## Coherence vs. Consistency

<table>
<thead>
<tr>
<th>Cache coherence</th>
<th>Memory consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deals with the ordering of operations to a <em>single</em> memory location.</td>
<td>Deals with the ordering of operations to <em>different</em> memory locations.</td>
</tr>
</tbody>
</table>
| Tackled by hardware  
  • using coherence protocols.  
  • Hw. alone guarantees correctness but with varying performance | Tackled by consistency models  
  • supported by hardware, but  
  • software must conform to the model. |
## Coherence vs. Consistency

<table>
<thead>
<tr>
<th>Cache coherence</th>
<th>Memory consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deals with the ordering of operations to a <em>single</em> memory location.</td>
<td>Deals with the ordering of operations to <em>different memory locations</em>.</td>
</tr>
<tr>
<td><strong>Tackled by hardware</strong></td>
<td><strong>Tackled by consistency models</strong></td>
</tr>
<tr>
<td>• using coherence protocols.</td>
<td>• supported by hardware, but</td>
</tr>
<tr>
<td>• Hw. alone guarantees correctness but with varying performance</td>
<td>• software must conform to the model.</td>
</tr>
<tr>
<td><strong>All protocols realize same abstraction</strong></td>
<td><strong>Models provide diff. abstractions</strong></td>
</tr>
<tr>
<td>• A program written for 1 protocol can run w/o change on any other.</td>
<td>• Compilers must be aware of the model (no reordering certain operations …).</td>
</tr>
<tr>
<td></td>
<td>• Programs must “be careful” in using shared variables.</td>
</tr>
</tbody>
</table>
Two Approaches to Consistency

• Sequential consistency
  – Multi-threaded codes for uniprocessors automatically run correctly
  – How? Every shared R/W completes globally in program order
  – Most intuitive but worst performance

• Relaxed consistency models
  – Multi-threaded codes for uniprocessor need to be ported to run correctly
  – Additional instruction (memory fence) to ensure global order between 2 operations
Cache Coherence

• Do we need caches?
  – Yes, to reduce average data access time.
  – Yes, to reduce bandwidth needed for bus/interconnect.

• Sufficient conditions for coherence:
  – Notation: Request_{proc}(data)
  – Write propagation:
    • Rd_i(X) must return the “latest” Wr_j(X)
  – Write serialization:
    • Wr_i(X) and Wr_j(X) are seen in the same order by everybody
      – e.g., if I see w2 after w1, you shouldn’t see w2 before w1
      – ➔ There must be a global ordering of memory operations to a single location
  – Is there a need for read serialization?
A Coherent Memory System: Intuition

- Uniprocessors
  - Coherence between I/O devices and processors
  - Infrequent, so software solutions work
    - Uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches

- But coherence problem much more critical in multiprocessors
  - Pervasive
  - Performance-critical
  - Necessitates a hardware solution

* Note that “latest write” is ambiguous.
  - Ultimately, what we care about is that any write is propagated everywhere in the same order.
  - Synchronization defines what “latest” means.
Summary

• Shared memory with caches raises the problem of cache coherence.
  – Writes to the same location must be seen in the same order everywhere.
• But this is not the only problem
  – Writes to *different* locations must also be kept in order if they are being depended upon for synchronizing tasks.
  – This is called the memory-consistency problem