• What can the programmer assume about the servicing of memory operations?
  ❖ For example, will they occur in program order?
  ❖ Why are these assumptions important?
The Uniprocessor Case

- What is the model of memory behavior?
  - Memory operations occur in program order → read returns the value of the last write in program order
- Semantics defined by sequential program order
  - Simple to reason about but over constrained
    - Really only need to honor control and data dependencies
  - Reality is that independent operations can execute in parallel
  - Optimizations preserve these semantics

Memory Consistency Issue

- What do you expect for the following codes?

  **Initial values**  
  A=0  
  B=0

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>
  | A=1; | while (Flag==0) {};
  | Flag = 1; | print A;               |

  Is it possible P2 prints A=0?

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1;</td>
<td>print B;</td>
<td></td>
</tr>
<tr>
<td>B=1;</td>
<td>print A;</td>
<td></td>
</tr>
</tbody>
</table>

  Is it possible P2 prints A=0, B=1?

*Source: Prof. Hsien Hsin Lee*
The Multiprocessor Case

- A memory consistency model →
  - A set of rules governing how the memory systems will process memory operations from multiple processors
  - Contract between the programmer and system
  - Determines what optimizations can be performed for correct programs

Importance

- Establishes the requirements for correct operation
- Correctness requirements govern implementation of system optimizations → ensure no violation of correctness requirements in pursuit of performance
- Correctness requirements govern programmer behavior → impact algorithm design
Maintain Program Ordering (SC)

Flag1 = Flag2 = 0

P1  P2

Flag1 = 1  Flag2 = 1
if (Flag2 == 0)  if (Flag1 == 0)
Enter Critical Section  Enter Critical Section

Caveat: implementation fine with uni-processor, but violate the ordering of the above

Source: Prof. Hsien Hsin Lee

Sequential Consistency

[Lamport] “A multiprocessor system is sequentially consistent if the result of any execution is the same as if 1) the operations of all processors were executed in some sequential order, and 2) the operations of each individual processor appear in this sequence in the order specified by the program”
Implications of Sequential Consistency

- Program order requirement
  - Note that memory systems may be parallel and that the network between processors and memories may re-order instructions

- Atomicity requirement
  - Follows from the global “interleaving” requirement
  - Informally, a write takes place instantaneously with respect to the ability of all other processors to read it


SC Example

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1</td>
<td>A=2</td>
<td></td>
<td></td>
<td>A=1</td>
<td>A=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T=A</td>
<td>Y=A</td>
<td></td>
<td></td>
<td>T=A</td>
<td>Y=A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U=A</td>
<td>Z=A</td>
<td></td>
<td></td>
<td>U=A</td>
<td>Z=A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T=1</td>
<td>Y=1</td>
<td></td>
<td></td>
<td>T=1</td>
<td>Y=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U=2</td>
<td>Z=2</td>
<td></td>
<td></td>
<td>U=2</td>
<td>Z=1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sequentially Consistent  Violating Sequential Consistency! (but possible in processor consistency model)

Source: Prof. Hsien Hsin Lee
Atomic and Instantaneous Update (SC)

A = B = 0

P1 P2 P3

A = 1

if (A==1)
  B = 1
  if (B==1)
    R1 = A

Caveat when an update is not atomic to all ...

Source: Prof. Hsien Hsin Lee

Atomic and Instantaneous Update (SC)

A = B = 0

P1 P2 P3

A = 1

if (A==1)
  B = 1
  if (B==1)
    R1 = A

Caches also make things complicated

P3 caches A and B

A=1 will not show up in P3 until P3 reads it in R1=A

Source: Prof. Hsien Hsin Lee
Program Ordering Issues

- Violation of program order requirement → can lead to incorrect parallel programs
  - Use of write buffers
    - Note: Does not violate data dependence in uniprocessor systems

- Ordering issues arise naturally in systems with caches

- Compiler re-ordering of instructions lead violations of sequential consistency

Maintain Program Ordering (SC)

- Dekker’s algorithm
- Only one processor is allowed to enter the CS

Flag1 = Flag2 = 0

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag1 = 1</td>
<td>Flag2 = 1</td>
</tr>
<tr>
<td>if (Flag2 == 0)</td>
<td>if (Flag1 == 0)</td>
</tr>
<tr>
<td>enter Critical Section</td>
<td>enter Critical Section</td>
</tr>
</tbody>
</table>

Caveat: implementation fine with uni-processor, but violate the ordering of the above

INCORRECT!!
BOTH ARE IN CRITICAL SECTION!

Source: Prof. Hsien Hsin Lee
Atomicity Issues in the Presence of Caches

- Consider the above example and an update based protocol
- Atomicity can be ensured by the following two conditions
  - Write completion: result of a write cannot be used until all copies have been updated/invalidated
    - A write must be atomic “system wide”
  - Writes to a location are serialized, i.e., all processors see writes to the same location in the same order

Caches Coherency vs. Consistency

- Coherence can be viewed as mechanism to propagate new values (equivalently invalidations) to cached copies
- Coherence requirements typically are
  - All writes to a variable are eventually visible to all processors
  - Serialization of writes to a variable
- Preceding insufficient for sequential consistency which requires serialization of all writes to all locations
Relaxed Memory Models

- Relaxation of memory requirements is achieved along two dimensions
  - Relaxing the program ordering requirement
  - Relaxing the atomicity requirement

- Key idea
  - Let reads and writes complete out of order
  - Use synchronization primitives to enforce ordering constraints
  - The program can behave “as if” it were a sequentially consistent execution

- Goal is increased performance from concurrency in memory operations

---

Relaxed Memory Models (cont.)

- Processor Consistency
  - Writes by any processor are seen by all processors in the order they were issued
  - For any variable, all processors see writes in the same order
  - Weaker than sequential consistency since the same ordering is not guaranteed to be seen by all processors
    - Write sequences from different processors can be interleaved (as viewed by other processors)
Relaxed Memory Models (cont.)

- **Weak Consistency**
  - Distinguish between data operations and synchronization operations
  - Synchronization operations are sequentially consistent
    - All processors see synchronization operations in the same order
  - When a synchronization operation is issued, the memory pipeline is flushed
    - All pending writes must complete before the synchronization operation executes

- **Release Consistency**
  - Increases overlap in memory operations restricted by the weak consistency model
    - Similarity with instruction issue and dependences?

---

Relaxed Memory Models

- **How to relax program order requirement?**
  - Load bypass store
  - Load bypass load
  - Store bypass store
  - Store bypass load

- **How to relax write atomicity requirement?**
  - Read others’ write early
  - Read own write early
Relaxed Consistency

- Processor Consistency
  - Used in P6
  - Write visibility could be in different orders of different processors (not guarantee write atomicity)
  - Allow loads to bypass independent stores in each individual processor
  - To achieve SC, explicit synchronization operations need to be substituted or inserted
    - Read-modify-write instructions
    - Memory fence instructions

```
F1=1  F2=1
A=1   A=2
R1=A  R3=A
R2=F2 R4=F1
```

R1=1; R3=1; R2=0; R4=0 is a possible outcome

Source: Prof. Hsien Hsin Lee
Processor Consistency

A = Flag = 0

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td>while (Flag==0);</td>
</tr>
<tr>
<td>Flag=1</td>
<td>Print A</td>
</tr>
</tbody>
</table>

Intuitive for event synchronization
“A” must be printed “1”

Source: Prof. Hsien Hsin Lee

Processor Consistency

A = B = 0

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (A==1)</td>
<td>if (B==1)</td>
<td></td>
</tr>
<tr>
<td>B =1</td>
<td>R1=A</td>
<td></td>
</tr>
</tbody>
</table>

B=1;R1=0 is a possible outcome
Since PC allows A=1 to be visible in P2 prior to P3

Source: Prof. Hsien Hsin Lee
Summary

• Consistency models are a set of rules that can be relied on by the programmer and compiler
• Consistency Models determine the system optimizations that are possible
  ❖ Overlapping of memory operations from multiple processors
• Many optimizations can violate consistency model semantics
  ❖ Leads to incorrect execution
• Consistency models are distinct from coherence
  ❖ The latter is concerned with updates/invalidations to a single shared variable
  ❖ The former is concerned with the behavior of memory references from multiple concurrent threads

Study Guide

• Be able to distinguish between types of consistency models
  ❖ From the perspective of definition
• Given a set of LD/SD references from multiple threads, be able to state the ordering and atomicity constraints on these LD/SD operations for a consistency model
  ❖ Can the follow sequence of operations happen if the memory system is sequentially consistent?
  ❖ Can the following sequence of operations happen if the memory system adheres to processor consistency?
  ❖ Will this optimization violate sequential consistency?
• Given a sequence of references taking place in a specific order does it satisfy the constraints of a specific model?
• Given the state of lines in multiple caches, what are the states of the cache lines after a given sequence of references
  ❖ For a snooping cache coherence protocol
  ❖ For a directory based protocol
• What is the performance impact of choices such as
  ❖ Invalidation vs. update protocols
  ❖ Line size
  ❖ Write back vs. write through
• Extend each protocol (i.e., extend the state machine) to use a new state: exclusive but unmodified