Overview: Memory Consistency

- the ordering of memory operations
- basic definitions; sequential consistency
- comparison with cache coherency
- relaxing memory consistency
  - write buffers
  - the total store ordering and processor consistency models
  - the partial store ordering model and write overlapping
  - weak and relaxed consistency models

Refs: Ch 8, pp 262-265, of Wilkinson & Allen; Shared Memory Consistency Models: A Tutorial
Example: The Ordering of Memory Operations

Consider the following (initially \texttt{flag1 = flag2 = 0}):  

\begin{align*}
\text{Process 0} & \quad \text{Process 1} \\
\text{flag1 = 1} & \quad \text{flag2 = 1} \\
\text{if (flag2 == 0)} & \quad \text{if (flag1 == 0)} \\
\text{print } "Hello"; & \quad \text{print } "World";
\end{align*}

What is (or can be) printed?
Time Ordered Events

```
Process 0
1 flag1 = 1
2 if (flag2 == 0) print "Hello"
3
4

Process 1
flag2 = 1
1
2 if (flag1 == 0) print "World"

Output: Hello
```

```
1 flag1 = 1
2
3 if (flag2 == 0) print "Hello"
4

Output: (Nothing Printed)
```

```
1
2 flag2 = 1
3
4 if (flag1 == 0) print "World"

Output: World
```

- **never** Hello and World?
- but what fundamental assumption are we making?
To write correct and efficient shared memory programs, programmers need a precise notion of how memory behaves with respect to read and write operations from multiple processors. (Adve and Gharachorloo)

Memory/cache coherency defines requirements for the observed behaviour of reads and writes to the same memory location

- i.e. ensures all processors have a consistent view of same address, i.e. agree on the order of operations
- this addresses problem of duplicated data in multiple caches / main memory

Memory consistency defines the behavior of reads and writes to different locations (as observed by other processors)

- i.e. when a write propagates to another processor, relative to its reads and writes

Sequential Consistency

- A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the 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 [Lamport, 79]

- Two aspects:
  - Maintaining program order among operations from individual processors
  - Maintaining a single sequential order among operations from all processors

- The latter aspect makes it appear as if a memory operation executes atomically or instantaneously with respect to other memory operations.
Programmer’s View of Sequential Consistency

- conceptually:
  - there is a single global memory and a switch that connects an arbitrary processor to memory at any time step
  - each processor issues memory operations in program order and the switch provides the global serialization among all memory operations
Cache Coherency and Sequential Consistency

- **one definition of** cache coherency:
  - a write is *eventually* made visible to all processors
  - writes to the *same* location appear to be seen in the same order by all processors

- **whereas** sequential consistency **requires**:
  - writes to *all* memory locations appear to be seen in the same order by all processors
  - write operations of a single processor appear to execute in program order

- **multiple caches and writes**
  - sequential consistency **requires** memory operations to appear atomic
    - but propagating changes to multiple cache copies is inherently non-atomic!
  - one option is to prohibit a read from returning a newly written quantity until all cached copies have acknowledged receipt of the *invalidation* or *update* messages
    - easier to ensure for the *invalidate* protocol
Motivation for Relaxed Consistency

- to gain performance:
  - hide latency of independent memory accesses with other operations
  - e.g. relaxing write to read and write to write allows writes to different locations to be pipelined or overlapped
  - relaxing write atomicity allows a read to return another processor's write before all cached copies have been updated

- recall that memory access to a cache-coherent system may involve much work!

- can relax either the program order or atomicity requirements (or both)
Possible Re-orderings

- we consider the program order requirements
- four types of memory operation orderings (within a program):
  - $W \rightarrow R$: write must complete before subsequent read (RAW)
  - $R \rightarrow R$: read must complete before subsequent read (RAR)
  - $R \rightarrow W$: read must complete before subsequent write (WAR)
  - $W \rightarrow W$: write must complete before subsequent write (WAW)
- Normally, different addresses are involved in the pair of operations
- note that sequential consistency preserves all 4
- relaxing these can give:
  - $W \rightarrow R$: e.g. the write buffer (aka store buffer)
  - everything: the weak and release consistency models
write buffer: (very common)

- the processor inserts writes into the write buffer and proceeds assuming it completes in due course
- subsequent reads bypass previous writes as long as the read address does not overlap with those in write buffer
- subsequent writes are processed by the buffer in order (no W→W relaxations)
Consistency Models Breaking the W→R Ordering

- **the total store ordering** and **processor consistency models** also relax W→R orderings (within a processor’s program order)
  - this allows a processor to move its reads to ahead of its own writes:
    - this can affect result of reads in parallel execution!
  - they still maintain W→W orders

  However, they also break atomicity constraints as well

- **total store ordering (TSO):**
  - a processor P can read the value of its write to A before it is seen by all other processors
  - reads by other processors cannot return the new value of A until the write to A is observed by all processors

- **processor consistency (PC):**
  - any processor can read the new value of A before the write is observed by all processors
Processor Consistency

- before a load is allowed to perform wrt. any processor, all previous load accesses must be completed wrt. everyone

- before a store is allowed to perform wrt. any processor, all previous load and store accesses must be completed wrt. everyone
Four Example Programs

Assume A and B are initialized to 0
Assume prints are loads

1
Thread 1 (on P1) Thread 2 (on P2)
A = 1;
flag = 1;

while (flag == 0);
print A;

2
Thread 1 (on P1)
A = 1;
B = 1;

Thread 2 (on P2)
print B;
print A;

3
Thread 1 (on P1) Thread 2 (on P2) Thread 3 (on P3)
A = 1;
while (A == 0);
B = 1;
print A;

4
Thread 1 (on P1) Thread 2 (on P2)
A = 1;
B = 1;

Do (all possible) results of execution match those of sequential consistency (SC)?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Store Ordering (TSO)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Processor Consistency (PC)</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

See http://15418.courses.cs.cmu.edu/spring2015/lecture/consistency
Consistency Models Breaking the $W \rightarrow R, W$ Orderings

- partial store ordering (PSO) is like TSO, but in addition it allows reordering of a processor’s write to different memory locations ($W \rightarrow W$)
- this can similarly affect parallel execution: execution may not match sequential consistency
- if initially $\text{flag} = A = 0$, P2 may observe the change to $\text{flag}$ before the change to $A$

<table>
<thead>
<tr>
<th>Thread 1 (on P1)</th>
<th>Thread 2 (on P2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 1$;</td>
<td>while ($\text{flag} == 0$);</td>
</tr>
<tr>
<td>$\text{flag} = 1$;</td>
<td>print $A$;</td>
</tr>
</tbody>
</table>

- in general, the technique of using a flag to signal a thread has written to a shared variable is unsafe here

  - are the lock and barrier algorithms from lectures 16-17 safe?
Breaking $W \rightarrow W$ Ordering: Overlapped Writes

**General (non-bus) interconnects with multiple memory modules:**

- Different memory operations issued by the same processor are serviced by different memory modules.
- Writes from $P1$ are injected into the memory system in program order, but they may complete out of program order.
- Many processors coalesce writes to the same cache line in a write buffer, and this can lead to similar effects.

(Adve and Gharachorloo DOI=10.1109/2.546611)
Allowing All Re-orderings: the Weak Consistency Model

- here, all four types of memory operation orderings (\(W \rightarrow R\), \(R \rightarrow R\), \(R \rightarrow W\) and \(W \rightarrow W\)) may be relaxed
- processors support must support a synch (or memory fence) instruction:
  - any memory accesses before the synch instruction must complete before it issues
  - memory accesses after the synch cannot begin until it completes
- relies on the programmer having used critical sections to control access to shared variables
  - within the critical section, no other processors can rely on that data structure being consistent until the critical section is exited
- we need to distinguish critical points when the program enters or leaves a critical section
  - i.e. distinguish standard load/stores from synchronization accesses
Weak Consistency Semantics

- before an ordinary load/store is allowed to perform wrt. any processor, all previous synch accesses must be completed wrt. everyone
- before a synch access is allowed to perform wrt. any processor, all previous ordinary load/store accesses must be completed wrt. everyone
- synch accesses are sequentially consistent wrt. one another
Allowing All Re-orderings: the Release Consistency Model

- Memory synchronization operations are divided into acquire and release pairs corresponding to entry and exit of a critical section.
- Before any ordinary load/store is allowed to perform wrt. any processor, all previous acquire accesses must be completed wrt. everyone.
- Before any release access is allowed to perform wrt. any processor, all previous ordinary load/store accesses must be completed wrt. everyone.
- Acquire/release accesses are processor consistent wrt. one another.
Example: Synchronization in Relaxed Models

- The hardware provides underlying instructions that are used to enforce consistency:
  - Fence or memory bar instructions
  - Normally, atomic instructions similarly synchronize memory

Different processors provide different types of fence instructions:

- Intel x86/x64: total store ordering
  - Provides sync. instructions if software requires a specific instruction ordering not guaranteed by the consistency model
    - `mm_lfence` (load fence): waits for all loads to complete
    - `mm_sfence` (store fence): waits for all stores to complete
    - `mm_mfence` (memory fence): waits for all memory operations to complete

- SPARC V9+: usually implement total store ordering also
  - Provided similar instructions, but also with load-load, load-store, store-load and store–store variants (ref)
  - E.g. memory copy via block load-store instructions program, `blkmemcpy.s`

- ARM processors: very relaxed consistency model
Summary: Relaxed Consistency

- motivation: obtain higher performance by reordering of memory operations (which is not allowed by sequential consistency)
- relaxed memory consistency issues arise from this reordering
  - this is irrespective of whether there are caches in the system!
- relaxed consistency models differ in which memory ordering constraints they ignore
- software complexity costs: programmer or compiler must (correctly) insert synchronization instructions to ensure the desired ordering
  - usually, these are encapsulated in system libraries that provide locks, barriers, etc
- final thoughts:
  - what consistency model best describes pthreads?
  - how about OpenMP, Java threads, ...