Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion
Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0
Flag2 = 0
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1
Flag2 = 0
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1
Flag2 = 0
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)

**critical section**

Process 2::
Flag2 = 1
If (Flag1 == 0)

**critical section**

Flag1 = 1
Flag2 = 0
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1
Flag2 = 1
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1
Flag2 = 1
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1
Flag2 = 1

Critical Section is Protected
Works the same if Process 2 runs first!
Process 2 enters its Critical Section
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag 1 = 1
If (Flag 2 == 0)
critical section

Process 2::
Flag 2 = 1
If (Flag 1 == 0)
critical section

Flag 1 = 1
Flag 2 = 0

Arbitrary interleaving of Processes
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
    critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
    critical section

Flag1 = 1
Flag2 = 1

Arbitrary interleaving of Processes
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Both processes can block but the critical section remains protected.

Deadlock can be fixed by extending the algorithm with turn-taking
Outline

- Concurrent Programming on a Uniprocessor
- The effect of optimizations on a Uniprocessor
- The effect of the same optimizations on a Multiprocessor without Sequential Consistency
- Methods for restoring Sequential Consistency
- Conclusion
Optimization: Write Buffer with Bypass

SpeedUp: Write takes 100 cycles, buffering takes 1 cycle. So Buffer and keep going.

Problem: Read from a Location with a buffered Write pending??
(Single Processor Case)
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0
Flag2 = 0

Flag1 = 1

Write Buffering
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0
Flag2 = 0

Flag1 = 1
Flag2 = 1

Write Buffering
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0
Flag2 = 0

Flag1 = 1
Flag2 = 1

Uh-Oh! Buffering
Optimization: Write Buffer with Bypass

SpeedUp: Write takes 100 cycles, buffering takes 1 cycle.

Rule: If a WRITE is issued, buffer it and keep executing

Unless: there is a READ from the same location (subsequent WRITEs don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

STALL!

Flag1 = 0
Flag2 = 0

Flag2 = 1
Flag1 = 1

Write Buffering

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent
WRITEs don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent WRITEs don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0
Flag2 = 0

Does this work for Multiprocessors??
Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0
Flag2 = 0

Does this work for Multiprocessors?
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent
WRITEs don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Multiprocessor Case

Flag1 = 0
Flag2 = 0

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent WRITEs don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1
Flag2 = 0
Flag2 = 1

Multiprocessor Case

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent
WRITEs don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Rule: If a WRITE is issued, buffer it and keep executing Unless: there is a READ from the same location (subsequent WRITES don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags INIT to 0

Process 1::
Flag1 = 1
If (Flag2 == 0) 
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0) 
critical section

Flag1 = 1
Flag2 = 1

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent WRITEs don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1
Flag2 = 0
Flag2 = 1

Multiprocessor Case

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent WRITEs don't matter), then wait for the WRITE to complete.
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1
Flag2 = 0

Multiprocessor Case

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent
WRITEs don't matter), then wait for the WRITE to complete.
What happens on a Processor stays on that Processor
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Rule: If a WRITE is issued, buffer it and keep executing
Unless: there is a READ from the same location (subsequent WRITEs don't matter), then wait for the WRITE to complete.

Processor 2 knows nothing about the write to Flag1, so has no reason to stall!

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0
Flag2 = 0
Flag2 = 1
A more general way to look at the Problem: Reordering of Reads and Writes (Loads and Stores).
Consider the Instructions in these processes.

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Simplify as:

WX
RY

WY
RX
There are 4! or 24 possible orderings.

If either WX<RX or WY<RY
Then the Critical Section is protected (Correct Behavior).
<table>
<thead>
<tr>
<th></th>
<th>WX</th>
<th>RY</th>
<th>WY</th>
<th>RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WX</td>
<td>RY</td>
<td>WY</td>
<td>RX</td>
</tr>
<tr>
<td>2</td>
<td>WX</td>
<td>RY</td>
<td>RX</td>
<td>WY</td>
</tr>
<tr>
<td>3</td>
<td>WX</td>
<td>WY</td>
<td>RY</td>
<td>RX</td>
</tr>
<tr>
<td>4</td>
<td>WX</td>
<td>RX</td>
<td>RY</td>
<td>WY</td>
</tr>
<tr>
<td>5</td>
<td>WX</td>
<td>WY</td>
<td>RX</td>
<td>RY</td>
</tr>
<tr>
<td>6</td>
<td>WX</td>
<td>RX</td>
<td>WY</td>
<td>RY</td>
</tr>
<tr>
<td>7</td>
<td>RY</td>
<td>WX</td>
<td>WY</td>
<td>RX</td>
</tr>
<tr>
<td>8</td>
<td>RY</td>
<td>WX</td>
<td>RX</td>
<td>WY</td>
</tr>
<tr>
<td>9</td>
<td>WY</td>
<td>WX</td>
<td>RY</td>
<td>RX</td>
</tr>
<tr>
<td>10</td>
<td>RX</td>
<td>WX</td>
<td>RY</td>
<td>WY</td>
</tr>
<tr>
<td>11</td>
<td>WY</td>
<td>WX</td>
<td>RX</td>
<td>RY</td>
</tr>
<tr>
<td>12</td>
<td>RX</td>
<td>WX</td>
<td>WY</td>
<td>RY</td>
</tr>
<tr>
<td>13</td>
<td>RY</td>
<td>WY</td>
<td>WX</td>
<td>RX</td>
</tr>
<tr>
<td>14</td>
<td>RY</td>
<td>RX</td>
<td>WX</td>
<td>WY</td>
</tr>
<tr>
<td>15</td>
<td>WY</td>
<td>RY</td>
<td>WX</td>
<td>RX</td>
</tr>
<tr>
<td>16</td>
<td>RX</td>
<td>RY</td>
<td>WX</td>
<td>WY</td>
</tr>
<tr>
<td>17</td>
<td>WY</td>
<td>RX</td>
<td>WX</td>
<td>RY</td>
</tr>
<tr>
<td>18</td>
<td>RX</td>
<td>WY</td>
<td>WX</td>
<td>RY</td>
</tr>
<tr>
<td>19</td>
<td>RY</td>
<td>WY</td>
<td>RX</td>
<td>WX</td>
</tr>
<tr>
<td>20</td>
<td>RY</td>
<td>RX</td>
<td>WY</td>
<td>WX</td>
</tr>
<tr>
<td>21</td>
<td>WY</td>
<td>RY</td>
<td>RX</td>
<td>WX</td>
</tr>
<tr>
<td>22</td>
<td>RX</td>
<td>RY</td>
<td>WY</td>
<td>WX</td>
</tr>
<tr>
<td>23</td>
<td>WY</td>
<td>RX</td>
<td>RY</td>
<td>WX</td>
</tr>
<tr>
<td>24</td>
<td>RX</td>
<td>WY</td>
<td>RY</td>
<td>WX</td>
</tr>
</tbody>
</table>

There are 4! or 24 possible orderings.

If either WX<RX or WY<RY
Then the Critical Section is protected (Correct Behavior)

18 of the 24 orderings are OK.
But the other 6 are trouble!
Consider another example...
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {};
LocalValue = Data

Memory Interconnect

Head = 0
Data = 0

Write By-Pass: General Interconnect to multiple memory modules means write arrival in memory is indeterminate.
Global DataInitialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {};
LocalValue = Data

Write By-Pass: General Interconnect to multiple memory modules means write arrival in memory is indeterminate.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {} 
LocalValue = Data

Write By-Pass: General Interconnect to multiple memory modules means write arrival in memory is indeterminate.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {;}
LocalValue = Data

Write By-Pass: General Interconnect to multiple memory modules means write arrival in memory is indeterminate.
Write By-Pass: General Interconnect to multiple memory modules means write arrival in memory is indeterminate.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {};
LocalValue = Data

Write By-Pass: General Interconnect to multiple memory modules means write arrival in memory is indeterminate.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {};
LocalValue = Data

Write By-Pass: General Interconnect to multiple memory modules means write arrival in memory is indeterminate.

Fix: Write must be acknowledged before another write (or read) from the same processor.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {};
LocalValue = Data

Non-Blocking Reads: Lockup-free Caches, speculative execution, dynamic scheduling allow execution to proceed past a Read.
Assume Writes are acknowledged.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {;}  
LocalValue = Data

Memory Interconnect

Head = 0
Data = 0

Non-Blocking Reads: Lockup-free Caches, speculative execution, dynamic scheduling allow execution to proceed past a Read.
Assume Writes are acknowledged.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {} 
LocalValue = Data

Memory Interconnect

Head = 0

Data = 0

Non-Blocking Reads: Lockup-free Caches, speculative execution, dynamic scheduling allow execution to proceed past a Read.
Assume Writes are acknowledged.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {;
LocalValue = Data (0)

Head = 0
Data = 0

Memory Interconnect

Non-Blocking Reads: Lockup-free Caches, speculative execution, dynamic scheduling allow execution to proceed past a Read.
Assume Writes are acknowledged.
Non-Blocking Reads: Lockup-free Caches, speculative execution, dynamic scheduling allow execution to proceed past a Read.
Assume Writes are acknowledged.
Global Data Initialized to 0

Process 1::
\[
\begin{align*}
\text{Data} &= 2000; \\
\text{Head} &= 1;
\end{align*}
\]

Process 2::
\[
\begin{align*}
\text{While (Head == 0) } &\{}; \\
\text{LocalValue} &= \text{Data} (0)
\end{align*}
\]

Memory Interconnect

Head = 1

Data = 2000

Non-Blocking Reads: Lockup-free Caches, speculative execution, dynamic scheduling allow execution to proceed past a Read.
Assume Writes are acknowledged.
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {}
LocalValue = Data (0)

Memory Interconnect

Head = 1
Data = 2000

Non-Blocking Reads: Lockup-free Caches, speculative execution, dynamic scheduling allow execution to proceed past a Read.
Assume Writes are acknowledged.
Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {};
LocalValue = Data (0)

Global Data Initialized to 0

Non-Blocking Reads: Lockup-free Caches, speculative execution, dynamic scheduling allow execution to proceed past a Read.
Assume Writes are acknowledged.
Let's reason about reordering of reads and writes again.
Consider the Instructions in these processes.

Process 1:
Data = 2000;
Head = 1;

Process 2:
While (Head == 0) {};
LocalValue = Data

Simplify as:

WX
WY

RY
RX
Correct behavior requires $WX<RX$, $WY<RY$. Program requires $WY<RX$.

$\Rightarrow$ 6 correct orders out of 24.
Correct behavior requires \( WX < RX, \) \( WY < RY. \) Program requires \( WY < RX. \)

\[ \Rightarrow 6 \text{ correct orders out of 24.} \]

Write Acknowledgment means \( WX < WY. \) Does that Help?

Disallows only 12 out of 24.

9 still incorrect!
Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion
Sequential Consistency for Multiprocessors

Why is it surprising that these code examples break on a multi-processor?

What ordering property are we assuming (incorrectly!) that multiprocessors support?

Are we assuming that they are sequentially consistent?
Sequential Consistency

Sequential Consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were interleaved arbitrarily. ...appears as if a memory operation executes atomically or instantaneously with respect to other memory operations

(Hennessy and Patterson, 4th ed.)
Understanding Ordering

- Program Order
- Compiled Order
- Interleaving Order
- Execution Order
Reordering

- Writes reach memory, and Reads see memory in an order different than that in the Program.
- Caused by Processor
- Caused by Multiprocessors (and Cache)
- Caused by Compilers
What Are the Choices?

If we want our results to be the same as those of a Sequentially Consistent Model. Do we:

- Enforce Sequential Consistency at the memory level?
- Use Coherent (Consistent) Cache?
- Or what?
Enforce Sequential Consistency?

Removes virtually all optimizations => Too Slow!
What Are the Choices?

If we want our results to be the same as those of a Sequentially Consistent Model. Do we:

- Enforce Sequential Consistency at the memory level?
- Use Coherent (Consistent) Caches?
- Or what?
Cache Coherence

- Multiple processors have a consistent view of memory (i.e. by using the MESI protocol)
- But this does not say *when* a processor must see a value updated by another processor.
- Cache coherency does not guarantee Sequential Consistency!
- Example: a write-through cache acts just like a write buffer with bypass.
What Are the Choices?

- Enforce Sequential Consistency? Too Slow!
- Use Coherent (Consistent) Caches? Won't help!
- What's left ??????
Involve the Programmer?
If you don't talk to your CPU about concurrency, who's going to??
What Are the Choices?

- Enforce Sequential Consistency? (Too Slow!)
- Use Coherent (Consistent) Cache? (Won't help)
- Provide Memory Barrier (Fence) Instructions?
Barrier Instructions

- Methods for overriding relaxations of the Sequential Consistency Model.
- Also known as a Safety Net.
- Example: A Fence would require buffered Writes to complete before allowing further execution.
- Not Cheap, but not often needed.
- Must be placed by the Programmer.
- Memory Consistency Model for Processor tells you how.
Consider the Instructions in these processes.

Process 1::
Flag1 = 1
>>Mem_Bar<<
If (Flag2 == 0)
critical section

Simplify as:

\[
\text{Fence: } WX < RY
\]

Process 2::
Flag2 = 1
>>Mem_Bar<<
If (Flag1 == 0)
critical section

Simplify as:

\[
\text{Fence: } WY < RX
\]
There are 4! or 24 possible orderings.

If either WX<RX or WY<RY
Then the Critical Section is protected (Correct Behavior)

18 of the 24 orderings are OK.
But the other 6 are trouble!

Enforce WX<RY and WY<RX.

Only 6 of the 18 good orderings are allowed OK.
But the 6 bad ones are forbidden!
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
>> Mem_Bar <<
If (Flag2 == 0)
critical section
Flag1 = 0

Process 2::
Flag2 = 1
>> Mem_Bar <<
If (Flag1 == 0)
critical section
Flag2 = 0

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
>> Mem_Bar <<
If (Flag2 == 0) critical section

Process 2::
Flag2 = 1
>> Mem_Bar <<
If (Flag1 == 0) critical section

Flag1 = 1
Flag2 = 0
Flag2 = 1

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1

>>Mem_Bar<<

If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1

>>Mem_Bar<<

If (Flag1 == 0)
critical section

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Dekker's Algorithm: Global Flags Init to 0

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Dekker's Algorithm: Global Flags Init to 0

Process 1::
Flag1 = 1
>>Mem_Bar<<
If (Flag2 == 0)
  critical section

Process 2::
Flag2 = 1
>>Mem_Bar<<
If (Flag1 == 0)
  critical section

Flag1 protects critical section when Process 2 continues at Mem_Bar.
Consider the Instructions in these processes.

Process 1::
Data = 2000;
>>Mem_Bar<<
Head = 1;

Process 2::
While (Head == 0) {};
>>Mem_Bar<<
LocalValue = Data

Simplify as:

**WX**
>>Fence<<
**WY**

**RY**
>>Fence<<
**RX**

Fence: WX < WY
Fence: RY < RX
Correct behavior requires $WX < RX$, $WY < RY$. Program requires $WY < RX$. $=> 6$ correct orders out of 24.

We can require $WX < WY$ and $RY < RX$. Is that enough? Program requires $WY < RX$. Thus, $WX < WY < RY < RX$; hence $WX < RX$ and $WY < RY$.

Only 2 of the 6 good orderings are allowed - But all 18 incorrect orderings are forbidden.
Global Data Initialized to 0

Process 1::
 Data = 2000;
>>Mem_Bar<<
Head = 1;

Process 2::
 While (Head == 0) {};
>>Mem_Bar<<
LocalValue = Data

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Process 1::
Data = 2000;
>>Mem_Bar<<
Head = 1;

Process 2::
While (Head == 0) {};
>>Mem_Bar<<
LocalValue = Data

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Global Data Initialized to 0

Process 1::
Data = 2000;
Head = 1;

Process 2::
While (Head == 0) {};
LocalValue = Data

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Global Data Initialized to 0

Process 1::
Data = 2000;
>>Mem_Bar<<
Head = 1;

Process 2::
While (Head == 0) {
>>Mem_Bar<<
LocalValue = Data
}

Memory Interconnect

Head = 1
Data = 2000

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Global Data Initialized to 0

Process 1::
Data = 2000;
>>Mem_Bar<<
Head = 1;

Process 2::
While (Head == 0) {};
>>Mem_Bar<<
LocalValue = Data

When Head reads as 1, Data will have the correct value when Process 2 continues at Mem_Bar.

Fence: Wait for pending I/O to complete before more I/O (includes cache updates).
Results appear in a Sequentially Consistent manner.
I've never heard of this. Is this for Real??
Memory Ordering in Modern Microprocessors, Part I

Linux provides a carefully chosen set of memory-barrier primitives, as follows:

- **smp_mb()**: "memory barrier" that orders both loads and stores. This means loads and stores preceding the memory barrier are committed to memory before any loads and stores following the memory barrier.

- **smp_rmb()**: "read memory barrier" that orders only loads.

- **smp_wmb()**: "write memory barrier" that orders only stores.
OK, I get it. So what's a Programmer supposed to do??
Words of Advice?

- “The difficult problem is identifying the ordering constraints that are necessary for correctness.”
- “…the programmer must still resort to reasoning with low level reordering optimizations to determine whether sufficient orders are enforced.”
- “…deep knowledge of each CPU's memory-consistency model can be helpful when debugging, to say nothing of writing architecture-specific code or synchronization primitives.”
Memory Consistency Models

- Explain what relaxations of Sequential Consistency are implemented.
- Explain what Barrier statements are available to avoid them.
- Provided for every processor (YMMV).
# Memory Consistency Models

<table>
<thead>
<tr>
<th></th>
<th>zSeries</th>
<th>x86 (x86-64 Store)</th>
<th>SPARC (SPARC PSO)</th>
<th>SPARC (SPARC RISC)</th>
<th>POWER</th>
<th>PA-RISC CPUs</th>
<th>IA64</th>
<th>AMD64</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads Reordered After Loads?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Loads Reordered After Stores?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Stores Reordered After Stores?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Stores Reordered After Loads?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Atomic Instructions Reordered With Loads?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Atomic Instructions Reordered With Stores?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Dependent Loads Reordered?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Incoherent Instruction Cache/Pipeline?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Programmer's View

- What does a programmer need to do?
- How do they know when to do it?
- Compilers & Libraries can help, but still need to use primitives in parallel programming (like in a kernel).
- Assuming the worst and synchronizing everything results in sequential consistency. Too slow, but a good way to debug.
How to Reason about Sequential Consistency

- Applies to parallel programs (Kernel!)
- Parallel Programming Language may provide the protection (DoAll loops).
- Language may have types to use.
- Distinguish Data and Sync regions.
- Library may provide primitives (Linux).
- How to know if you need synchronization?
Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion
Conclusion

- Parallel programming on a Multiprocessor that relaxes the Sequentially Consistent Model presents new challenges.
- Know the memory consistency models for the processors you use.
- Use barrier (fence) instructions to allow optimizations while protecting your code.
- Simple examples were used, there are others much more subtle. The fix is basically the same.
Conclusion

- Parallel programming on a Multiprocessor that relaxes the Sequentially Consistent Model presents new challenges.
- **Know the memory consistency models for the processors you use.**
- Use barrier (fence) instructions to allow optimizations while protecting your code.
- Simple examples were used, there are others much more subtle. The fix is basically the same.
Conclusion

- Parallel programming on a Multiprocessor that relaxes the Sequentially Consistent Model presents new challenges.
- Know the memory consistency models for the processors you use.
- Use barrier (fence) instructions to allow optimizations while protecting your code.
- Simple examples were used, there are others much more subtle. The fix is basically the same.
Conclusion

- Parallel programming on a Multiprocessor that relaxes the Sequentially Consistent Model presents new challenges.
- Know the memory consistency models for the processors you use.
- Use barrier (fence) instructions to allow optimizations while protecting your code.
- Simple examples were used, there are others much more subtle. The fix is basically the same.
References

- Shared Memory Consistency Models: A Tutorial By Sarita Adve & Kourosh Gharachorloo
- Vince Shuster Presentation for CS533, Winter, 2010
- Memory Ordering in Modern Microprocessors, Part I, Paul E. McKenney, Linux Journal, June, 2005
- Computer Architecture, Hennessy and Patterson, 4th Ed., 2007