Focus

- Given a collection of threads, how should they communicate information among one another?
- Using message-passing, they communicate through messages, information that is directed from one thread to another.
  - Sometimes the recipient may be anonymous
    - channel-based communication (CML)
  - Typically, the recipient is known
- What should the sender do after the message is sent?
  - wait until the recipient acknowledges receipt (synchronous)
  - proceed regardless (asynchronous)
Processor Models

• Processor architectures support the execution of multiple concurrent operations.

• When should an operation performed by one processor be available to another?

• Sequential consistency:
  - Every program executes instructions in sequential order.
  - All shared loads and stores to memory must occur before any other access.
Architecture

shared memory

cache

processor
Example

Initially all pointers = null, all integers = 0.
P1
while (there are more tasks) {
    Task = GetFromFreeList();
    Task -> Data = ...;
    insert Task in task queue
}
Head = head of task queue;
P2, P3, ..., Pn
while (MyTask == null) {
    Begin Critical Section
    if (Head != null) {
        MyTask = Head;
        Head = Head -> Next;
    }
    End Critical Section
}
... = MyTask -> Data;

What value can a read return?
Sequential consistency

- Maintain program order among operations from individual processors
- maintain a single execution order among operations from all processors
  - memory operations execute atomically

Initially Flag1 = Flag2 = 0
Initially A = B = 0

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag1 = 1</td>
<td>if (Flag2 == 0)</td>
<td>Flag2 = 1</td>
<td>A = 1</td>
<td>if (Flag1 == 0)</td>
<td>critical section</td>
</tr>
<tr>
<td></td>
<td>if (Flag2 == 0)</td>
<td>if (Flag1 == 0)</td>
<td></td>
<td>if (A == 1)</td>
<td>B = 1</td>
</tr>
</tbody>
</table>

(a) (b)

CS390C: Principles of Concurrency and Parallelism
Implementation

• Processors must ensure its previous memory operation completes before proceeding with the next one.
  − Memory must provide explicit acknowledgement that write has completed.
  − Caches must invalidate or update all cached copies

• Writes to the same location be made visible to all processors in the same order
  − serialized

• Value of a write not be returned until all invalidates or updates acknowledged.
Relaxed Memory Models

Relaxation

- Relax Write to Read program order
- Relax Write to Write program order
- Relax Read to Read and Read to Write program orders
- Read others’ write early
- Read own write early
How should memory operations be ordered?

- Assume coherent write-back caches.
- Operations that write to memory have a total order if all processors agree on their order of execution.
- Case study: Intel 64 memory ordering:
  - what are the rules that dictate how instructions can be reordered?
## Loads and stores seen in program order

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [ _x], 1 // M1</td>
<td>mov r1,[_y] // M3</td>
</tr>
<tr>
<td>mov [ _y], 1 // M2</td>
<td>mov r2, [ _x] // M4</td>
</tr>
</tbody>
</table>

Initially x == y == 0

r1 == 1 and r2 == 0 is not allowed

Stores M1 and M2 cannot be reordered at processor 0. Loads M3 and M4 cannot be reordered at processor 1.

M1 must appear to execute before M4, implying r2 == 1
### Stores not reordered with older loads

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov r1, [ _x]  // M1</td>
<td>mov r2, [ _y]  // M3</td>
</tr>
<tr>
<td>mov [ _y], 1   // M2</td>
<td>mov [ _x], 1   // M4</td>
</tr>
</tbody>
</table>

Initially x == y == 0

r1 == 1 and r2 == 1 is not allowed

Stores not reordered with older loads
Loads may be reordered with older stores to different locations

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [ _x], 1    // M1</td>
<td>mov [ _y], 1    // M3</td>
</tr>
<tr>
<td>mov r1, [ _y]  // M2</td>
<td>mov r2, [ _x]  // M4</td>
</tr>
</tbody>
</table>

Initially $x == y == 0$

r1 == 0 and r2 == 0 is allowed
### Loads not reordered with older stores to the same location

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [x], 1  // M1</td>
<td>mov [y], 1  // M3</td>
</tr>
<tr>
<td>mov r1, [x]  // M2</td>
<td>mov r2, [y]  // M4</td>
</tr>
</tbody>
</table>

Initially x == y == 0

Must have r1 == 1 and r2 == 1
Stores by two processors can be seen in different order

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [ _x], 1 // M1</td>
<td>mov [ _y], 1 // M4</td>
</tr>
<tr>
<td>mov r1, [ _x] // M2</td>
<td>mov r3, [ _y] // M5</td>
</tr>
<tr>
<td>mov r2, [ _y] // M3</td>
<td>mov r4, [ _x] // M6</td>
</tr>
</tbody>
</table>

Initially x == y == 0

r2 == 0 and r4 == 0 is allowed

Due to store-buffer forwarding
## Stores are transitively visible

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov [_x], 1 // M1</code></td>
<td><code>mov r1, [_x] // M2</code></td>
<td><code>mov r2, [_y] // M4</code></td>
</tr>
<tr>
<td><code>mov [_y], 1 // M3</code></td>
<td><code>mov r3, [_x] // M5</code></td>
<td></td>
</tr>
</tbody>
</table>

Initially `x == y == 0`

`r1 == 1, r2 == 1, r3 == 0` is not allowed

Stores that are causally related appear to execute in an order consistent with that relation
Total order to stores of the same location

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov  [ _x], 1       // M1</td>
<td>mov  [ _x], 2       // M2</td>
<td>mov  r1, [ _x]      // M3</td>
<td>mov  r3, [ _x]      // M5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov  r2, [ _x]      // M4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov  r4, [ _x]      // M6</td>
<td></td>
</tr>
</tbody>
</table>

Initially x == 0

r1 == 1, r2 == 2, r3 == 2, r4 == 1 is not allowed

Two stores to the same location (even by different processors) must appear to all processors to execute in the same order.
Locked instructions have a total order

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchg [ _x], r1 // M1</td>
<td>xchg [ _y], r2 // M2</td>
<td>mov r3, [ _x] //M3</td>
<td>mov r5, [ _y] //M5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov r4, [ _y] //M4</td>
<td>mov r6, [ _x] //M6</td>
</tr>
</tbody>
</table>

Initially x == y == 0, r1 == r2 == 1

r3 == 1, r4 == 0, r5 == 1, r6 == 0 is not allowed
## Loads and stores not reordered with respect to locks

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchg [ _x], r1 // M1</td>
<td>xchg [ _y], r3 // M3</td>
</tr>
<tr>
<td>mov r2, [ _y] // M2</td>
<td>mov r4, [ _x] // M4</td>
</tr>
</tbody>
</table>

Initially x == y == 0, r1 == r3 == 1

r2 == 0 and r4 == 0 is not allowed
Language Memory Models

- Many programming languages do not deal with concurrency, instead a library provides a set of API calls.
- This is brittle in the presence of optimizing compilers and modern architectures.
- Java is the first programming language to specify a memory model that must be enforced by the compiler.
- The memory model gives semantics to concurrent programs, a must if we want to program multicores.
Motivation: Double Checked Locking

- Double checked locking is an idiom that tries to avoid paying the cost of synchronization when not needed
  - Important for lazy initialization
- The prototypical example:

```java
// Single threaded version
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }
    // other functions and members...
}
```
Broken Double Checked Locking

- This avoids the cost of synchronization when helper is already initialized:

```java
class Foo {
    private Helper helper;
    Helper getHelper() {
        if (helper == null)
            synchronized(this) {
                if (helper == null) helper = new Helper();
            }
        return helper;
    }
}
```

- Problems:

  - The writes that initialize the Helper object and the write to the helper field can be out of order. A thread invoking getHelper() could see a non-null reference to a helper object, but see the default values for its fields, rather than the values set in the constructor.

  - If the compiler inlines the call to the constructor, then the writes that initialize the object and the write to the helper field can be freely reordered.

  - Even if the compiler does not reorder, on a multiprocessor the processor or memory system may reorder those writes, as perceived by a thread running on another processor.
More on why it’s broken

- **A test case showing that it doesn't work** (by Paul Jakubik). When run on a system using the Symantec JIT:

  ```java
  singletons[i].reference = new Singleton();
  ```

to the following (note that the Symantec JIT using a handle-based object allocation system).

```
0206106A  mov          eax,0F97E78h
0206106F  call         01F6B210  ; allocate space for
                        ; Singleton, return result in eax
02061074  mov          dword ptr [ebp],eax
                        ; EBP is &singletons[i].reference
                        ; store the unconstructed object here.
02061077  mov          ecx,dword ptr [eax]
                        ; dereference the handle to
                        ; get the raw pointer
02061079  mov          dword ptr [ecx],100h
0206107F  mov          dword ptr [ecx+4],200h
02061086  mov          dword ptr [ecx+8],400h
0206108D  mov          dword ptr [ecx+0Ch],0F84030h
```

The assignment to `singletons[i].reference` is performed before the constructor for `Singleton` is called. This is completely legal under the existing Java memory model, and also legal in C and C++ (since neither of them have a memory model).
The Java™ Memory Model: the building block of concurrency

Jeremy Manson, Purdue University
William Pugh, Univ. of Maryland
http://www.cs.umd.edu/~pugh/java/memoryModel/

TS-1630
Synchronization is needed for Blocking and Visibility

- Synchronization isn’t just about mutual exclusion and blocking
- It also regulates when other threads *must* see writes by other threads
  - When writes become visible
- Without synchronization, compiler and processor are allowed to reorder memory accesses in ways that may surprise you
  - And break your code
Don’t Try To Be Too Clever

- People worry about the cost of synchronization
  - Try to devise schemes to communicate between threads without using synchronization
    - locks, volatiles, or other concurrency abstractions
- Nearly impossible to do correctly
  - Inter-thread communication without synchronization is not intuitive
Quiz Time

x = y = 0

Thread 1

x = 1

j = y

Thread 2

y = 1

i = x

Can this result in i = 0 and j = 0?
Answer: Yes!

How can $i = 0$ and $j = 0$?

$\begin{align*}
x &= 1 \\
i &= x
\end{align*}$

$\begin{align*}
y &= 1 \\
j &= y
\end{align*}$
How Can This Happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized to global memory
- The memory model is designed to allow aggressive optimization
  - including optimizations no one has implemented yet
- Good for performance
  - bad for your intuition about insufficiently synchronized code
When Are Actions Visible to Other Threads?

Thread 1:
- \texttt{ref1.x = 1}
- \texttt{lock M}
- \texttt{glo = ref1}
- \texttt{unlock M}

Everything before an unlock (release)

Is visible to everything after a later lock (acquire) on the same Object

Thread 2:
- \texttt{lock M}
- \texttt{ref2 = glo}
- \texttt{unlock M}
- \texttt{j = ref2.x}
Release and Acquire

- All memory accesses before a release
  - are ordered before and visible to
  - any memory accesses after a matching acquire

- Unlocking a monitor/lock is a release
  - that is acquired by any following lock of that
    monitor/lock
Happens-before ordering

- A release and a matching later acquire establish a *happens-before* ordering

- execution order within a thread also establishes a happens-before order

- happens-before order is transitive
Data race

- If there are two accesses to a memory location,
  - at least one of those accesses is a write, and
  - the memory location isn’t volatile, then
- the accesses *must* be ordered by happens-before

- Violate this, and you may need a PhD to figure out what your program can do
  - not as bad/unspecified as a buffer overflow in C
Volatile fields

- A field marked as **volatile** will be treated specially by the compiler/JIT
- read/writes go directly to memory and are never cached in registers
- volatile long/double are atomic
- volatile operations can’t be reordered by the compiler

```java
class Animator implements Runnable {
    private volatile boolean stop = false;
    public void stop() { stop = true; }
    public void run() {
        while (!stop)
            oneStep();
        try { Thread.sleep(100); } ...;
    }
    private void oneStep() { /*...*/ }
}
```
Volatile fields

- Volatile fields induce happens-before edges, similar to locking
- Incrementing a volatile is not atomic
  - if threads try to increment a volatile at the same time, an update might get lost
- volatile reads are very cheap; volatile writes cheaper than synchronization
- atomic operations require compare and swap; provided in JSR-166 (concurrency utils)
Correct Double Checked Locking

- This avoids the cost of synchronization when helper is already initialized:

```java
class Foo {
    private volatile Helper helper;
    Helper getHelper() {
        if (helper == null)
            synchronized(this) {
                if (helper == null) helper = new Helper();
            }
        return helper;
    }
}
```
Concurrencey Utilities and Atomics

- The JSR-166 (designed by Doug Lea at SUNY Oswego) has introduced a rich family of API for concurrent programming