Revising the Java Thread/Memory Model

See

http://www.cs.umd.edu/~pugh/java/memoryModel
for more information
Audience

• This will be an advanced talk
• Helpful if
  – you’ve been aware of the discussion,
  – have implemented a JVM,
  – know what sequential consistency is, and that most processors don’t support it, or
  – have read Doug Lea’s book
• The easy version of this talk is Thursday, 1:30, Hall C.
Java Thread Specification

• Chapter 17 of the Java Language Spec
  – Chapter 8 of the Virtual Machine Spec
• Very, very hard to understand
  – not even the authors understood it
  – doubtful that anyone entirely understands it
  – has subtle implications
    • that forbid standard compiler optimizations
  – all existing JVMs violate the specification
    • some parts should be violated
Revising the Thread Spec

• Work is underway to consider revising the Java Thread Spec
  – http://www.cs.umd.edu/~pugh/java/memoryModel

• Goals
  – Clear and easy to understand
  – Foster reliable multithreaded code
  – Allow for high performance JVMs

• Will effect JVMs
  – and badly written existing code
    • including parts of Sun’s JDK
When’s the JSR?

• Very hard and technical problems need to be solved
  – formal specification is difficult
  – not appropriate for JSR process
• Once we get technically solid proposals
  – we will start JSR process
  – aiming to start this fall
• Will miss Merlin cutoff
• Workshop at OOPSLA
Proposed Changes

• Make it clear
• Allow standard compiler optimizations
• Remove corner cases of synchronization
  – enable additional compiler optimizations
• Strengthen volatile
  – make easier to use
• Strengthen final
  – Enable compiler optimizations
  – Fix security concerns
VM Safety

- Type safety
- Not-out-of-thin-air safety
  - (except for longs and doubles)
- No new VM exceptions
- Only thing lack of synchronization can do is produce surprising values for getfields/getstatics/array loads
VM Safety implications

• Problems on SMPs with weak memory models

• Could see uninitialized objects created by another thread
  – need to initialize memory during GC
  – worry about seeing null vptr
  – worry about seeing zero array length

• Class loading and initialization issues
Weird Behavior of Improperly Synchronized Code

\[ x = y = 0 \]

Thread 1
\[ x = 1 \]
\[ j = y \]

Thread 2
\[ y = 1 \]
\[ i = x \]

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

Thread 1

\[ x = y = 0 \]

\[ x = 1 \]

\[ j = y \]

Thread 2

\[ y = 1 \]

\[ i = x \]

How can \( i = 0 \) and \( j = 0 \)?
How Can This Happen?

• Compiler can reorder statements
  – or keep values in registers
• Processor can reorder them
• On multi-processor, values not synchronized in global memory

• Must use synchronization to enforce visibility and ordering
  – as well as mutual exclusion
Synchronization

• Synchronization on thread local objects
  – e.g., synchronized(new Object()) {}
  – is not a no-op under current semantics
  – but it isn’t a memory barrier

• Proposal: make it a no-op
  – and allow other compiler optimizations

• Programming model is release consistency
When Are Actions Visible to Other Threads?

Thread 1

\[ x = 1 \]

unlock M

Thread 2

lock M

\[ i = x \]
New Optimizations Allowed

• Turning synchronizations into no-ops
  – locks on objects that aren’t ever locked by any other threads
  – reentrant locks
  – enclosed locks

• Lock coarsening
  – merging two calls to synchronized methods on same object
    • need to be careful about starvation issues
Doesn’t work under either existing or proposed semantics

class Service { // DO NOT USE
    Parser parser = null;
    Parser getParser() {
        if (parser == null)
            synchronized(this) {
                if (parser == null)
                    parser = new Parser();
            }
        return parser;
    }
}
Existing Semantics of Volatile

• No compiler optimizations
  – Can’t hoist read out of loop
  – reads/writes go directly to memory

• Reads/writes of volatile are sequentially consistent
  – can not be reordered
  – but access to volatile and non-volatile variables can be reordered

• Reads/writes of long/doubles are atomic
Existing Volatile Compliance

• Very poor
  – some JVMs completely ignore volatile
• No one enforces sequential consistency
• Atomic longs/doubles isn’t enforced on most

• New compliance tests will likely be rolled out soon
### Volatile Compliance

<table>
<thead>
<tr>
<th></th>
<th>No Compiler Optimizations</th>
<th>Sequential Consistency</th>
<th>Atomic Longs/Doubles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solaris JDK 1.2.2 EVM</strong></td>
<td>Pass</td>
<td>Fail</td>
<td>Pass</td>
</tr>
<tr>
<td><strong>Solaris JDK 1.3.0 beta Hotspot Client</strong></td>
<td>Fail</td>
<td>Fail</td>
<td>Fail</td>
</tr>
<tr>
<td><strong>Windows JDK 1.3.0 Hotspot Client</strong></td>
<td>Fail</td>
<td>Fail</td>
<td>Fail</td>
</tr>
<tr>
<td><strong>Solaris JDK 1.3.0 beta Hotspot Server</strong></td>
<td>Pass</td>
<td>Fail</td>
<td>Fail</td>
</tr>
<tr>
<td><strong>Windows JDK 1.3.0 Hotspot Server</strong></td>
<td>Pass</td>
<td>Fail</td>
<td>Fail</td>
</tr>
<tr>
<td><strong>Windows IBM JDK 1.1.8</strong></td>
<td>Pass</td>
<td>Fail</td>
<td>Fail</td>
</tr>
</tbody>
</table>
Need for volatile

```java
int answer = 0;
boolean ready = false;

answer = 42;
ready = true;

if (ready)
    System.out.println(answer);
```

Can print 0
Need for volatile

```
volatile int answer = 0;
volatile boolean ready = false;
```

**start threads**

```
answer = 42;
ready = true;
```

```
if (ready)
    System.out.println(answer);
```

**Must not print 0**
Proposed New Semantics for Volatile

- Write to a volatile acts as a release
- Read of a volatile acts as an acquire
- If a thread reads a volatile
  - all writes done by any other thread,
  - before earlier writes to the same volatile,
  - are guaranteed to be visible
New semantics for volatile

```java
int answer = 0;
volatile boolean ready = false;

answer = 42;
ready = true;

if (ready)
    System.out.println(answer);
```

Existing semantics: can print 0
Proposed semantics: must not print 0
When Are Actions Visible to Other Threads?

Thread 1

```
answer = 42
ready = true
```

anything done by thread 1, before writing `ready`

must be visible to any operations in thread 2 that occur after readying `ready`

Thread 2

```
if (ready)
println(answer)
```
Naïve Implementation of Volatile

- On SMP with weak memory model (Alpha)
  – Membar before & after each volatile write
  – Membar after each volatile read
- On SMP with TSO (e.g. Sparc)
  – Membar after each volatile write
- On IA-64
  – use ld.acq and st.rel for volatile fields
  – also, memory barrier after each volatile write
Implementation Cost of Proposed Change in Semantics

- Naïve implementation handles new semantics
  - unclear if only enforcing only existing semantics would incur fewer memory barriers
- New semantics will prohibit some compiler optimizations
  - reading a volatile will force all values cached in registers to be reloaded
Volatile Summary

• These semantics make volatile rather heavy weight
  – may not be cheaper than synchronization

• Few programmers will use all these features
  – Do we really need sequential consistency, on top of acquire/release semantics?

• But it is simple and easy to understand
  – more likely to be used correctly
Immutable Objects

• Many Java classes represent immutable objects
  – e.g., String

• Creates many serious security holes if Strings are not truly immutable
  – probably other classes as well
  – should do this in String implementation, rather than in all uses of String
Strings aren’t immutable

just because thread 2 sees new value for `Global.s`
doesn’t mean it sees all writes done by thread 1
before store to `Global.s`

Compiler, processor or memory system can reorder these writes

Symantic JIT will do it
Why aren’t Strings immutable?

• A String object is initialized to have default values for its fields
• *then* the fields are set in the constructor
• Thread 1 could create a String object
• pass it to Thread 2
• which calls a sensitive routine
• which sees the fields change from their default values to their final values
Making String immutable

• Could make String methods synchronized
  – most programmers don’t think methods for immutable objects need to be synchronized
  – synchronization would slow down String methods on all platforms
    • only needs to be synchronized on SMP’s with weak memory models
    • doesn’t need synchronization on SPARC or MAJC(?) SMP’s
Final = Immutable?

• Existing Java memory model doesn’t mention final
  – no special semantics

• Would be nice if compiler could treat final fields as constant
  – Don’t have to reload at memory barrier
  – Don’t have to reload over unknown function call
Existing semantics require that final fields need to be reloaded at synchronization points

class A extends Thread {
    final int x;
    A() {
        synchronized(this) {
            start();
            sleep(10);
        }
        x = 42;
    }
}

public void run() {
    int i,j;
    i = x;
    synchronized(this) {
        j = x;
    }
    System.out.println(i+j);
}

Must *not* print 0
Proposed Semantics for Final

• Read of a final field always sees the value set in constructor
  – If,
    • a final field is read before set
      – (by the constructing thread)
    • or, a reference to the object becomes visible to another thread before object is constructed
      • semantics are ugly
  • Can assume final fields never change
  • Makes string immutable?
Problems

• JNI code can change final fields
  – setIn, setOut, setErr
  – **Propose to remove this ability**
  – (reflection appears to be safe)

• Objects that escape their constructor before final fields are set
  – Base class “registers” object, derived class has final fields

• Doesn’t suffice to make strings immutable
Doesn’t make Strings immutable

- No way for elements of an array to be final
- For Strings, have to see final values for elements of character array
- So…
  - Read of final field is treated as a weak acquire
    - matching a release done when object is constructed
  - weak in that it only effects things dependent on value read
    - no compiler impact
Visibility enforced by final field \texttt{a}

All actions done before completion of constructor must be visible to any action that is data dependent on the read of a final field set in that constructor.

\begin{center}
\begin{tikzpicture}
    \node[draw] (a) {\texttt{this.a = new int[5]}};  \node[] (b) at (0,-1) {\texttt{this.a[0] = 42}};  \node[draw] (c) at (0,-2) {end constructor};  \node[draw] (d) at (0,-3) {\texttt{Foo.b = this}};  \node[draw] (e) at (3,-2) {\texttt{Foo.t = Foo.b}};  \node[draw] (f) at (3,-3) {\texttt{int[]} \texttt{tmp = t.a}};  \node[draw] (g) at (3,-4) {\texttt{... = tmp[0]}};  \node[draw] (h) at (3,-5) {\texttt{... = Foo.x}};

    \path[->] (a) edge (b) (b) edge (c) (c) edge (d) (d) edge (e) (e) edge (f) (f) edge (g) (g) edge (h);
\end{tikzpicture}
\end{center}
Contrast with volatile

Actions done before assignment to volatile field

must be visible to any action after the read

this.a = new int[5]
end constructor

this.a[0] = 42

Foo.x++

Foo.t = Foo.b

int[] tmp = t.a

... = tmp[0]

Foo.b = this

... = Foo.x
Data dependence is transitive

```java
Foo x++

this.a = new int[5][5]

this.a[0][0] = 42

end constructor

Foo b = this

Foo t = Foo.b

int[][] tmp = t.a

int[] tmp2 = tmp[0]

... = Foo.x

... = tmp2[0]
```
Thread Communication

• All forms of inter-thread communication force writes to be visible
  – interrupt
  – start/join
  – isAlive

• Sleep and yield have no effect on visibility
  – will cause problems for broken programs
  – but difficult/impossible to specify semantics of visibility for sleep
finalization

• Loosing the last reference to an object is an asynchronous signal to another thread to run the finalizer
  – which writes, done before loosing the last ref
  – are visible to the finalizer?
• Proposal: only writes to the object being finalized
  – need synchronization to see other writes
• Unsynchronized finalizers are dubious