Overview of Java Memory Model

Joe Hendrix

Singleton Example

Consider the following code:

```
class Foo {
    int x;
    Foo() { x = 2; }

    static Foo singleton = null;
    static Foo getFoo() {
        if (singleton == null)
            synchronize(Foo.class) {
                if (singleton == null)
                    singleton = new Foo();
            }
    }
}
```

Memory Models

- A *Memory Model* controls what values a read from a shared variable is allowed to see.
- It does not affect variables that are accessed by only a single thread, nor does it affect variables that are properly guarded by synchronization primitives.

Sequential Consistency

Sequential Consistency is the memory model where there is a total order over the actions in a trace, and each read sees the value of the most recent write in the total order.

Thread 1	*Thread 2*
L1 : R1 = X	L3 : R2 = Y
L2 : Y = R1	L4 : X = R2

An Execution

```
L1: read (0, init)
L2: write 0
                    L3: read (0, L2)
                    L4: write 0
```

Relaxed Memory Models

- The current trend in architecture is towards *Relaxed Memory Models*. These are models that allow reads to return values other than the values allowed by sequentially consistency.
- From an architecture perspective, relaxed models are currently mainly a way to save money. By adding extra hardware, one can achieve similar performance with sequential consistency.
- The problem is that optimizing compilers are greatly restricted if they are required to maintain sequential consistency.

Java Memory Model

Specifying a memory model for Java is particularly hard.

- Java is suppossed to run on many different processor types with different relaxed models. It needs to be relaxed enough that it runs without much performance overhead on all those architectures.
- Java is geared as a mainstream programming language. It needs to be designed so that business system programmers can understand it.
- The main two proposed models are both very relaxed models. They basically are designed to allow as many optimizations as possible.

Happens Before

$i\ hb\ j$ if and only if:

- $i\ hb\ k$ and $k\ hb\ j$ for some k.
- $thread(i) = thread(j)$ and i is before j in program order.
- i is a *synchronization* write and j is a synchronization that returns the value writen by i.

Thread 1	*Thread 2*
L1 : R1 = X	L3 : R2 = Y
L2 : Y = R1	L4 : X = R2

In the ordering (L1, L2, L3, L4), L1 hb L2 and L3 hb L4. Only if Y was declared volatile, would we have L2 hb L3.

Consistency Ordering Requirements

- First, a memory model called *Consistency* will be introduced. It is a very relaxed model that allows a superset of the executions allowed by both Java memory models.
- In Consistency, a read r is *allowed* to observe a write w to a variable v if:
  - r is not ordered before w (ie. $\neg(r\ hb\ w)$).
  - There is no write w' to v such that $w\ hb\ w'\ hb\ r$.
- Consistency is a relaxed enough model to allow the optimizations that architects and compiler writers want. Unfortunately, it also allows executions that are deemed too counterintuitive.

Java Memory Model Goals

- There are two proposed Java Memory Models:
  - MP Model by Jeremy Manson and William Pugh of UMD
  - Adve Model by Sarita Adve of UIUC.
- It is an open question on whether they are equivalent, but they share the same two requirements (above the requirements of consistency):
  - Do not allow values to appear *Out of thin air*
  - If all sequentially consistent executions are properly synchronized, then the only allowed executions are the sequentially consistent ones.

Simple Reordering Example

Thread 1	*Thread 2*
L1 : R1 = X	L3 : R2 = Y
L2 : Y = 1	L4 : X = 1

Since L1 and L2 are independent in Thread 1, a unithread compiler would normally be able to reorder them. But this leads to the sequentially consistent execution:

L2: write 1	
	L3: read (1, L2)
	L4: write 1
L1: read (1, L4)	

Out-of-Thin-Air Example

- Going back to the first program:

Thread 1	*Thread 2*
`L1: R1 = X`	`L3: R2 = Y`
`L2: Y = R1`	`L4: X = R2`

- This allows the value of 7 to appear in the execution:

```
L1: read (7, L4)
L2: write 7
                    L3: read (7, L2)
                    L4: write 7
```

- Any Java Memory Model needs to avoid these Out-of-Thin-Air examples.

Datarace

- A datarace occurs when there are two actions in an execution that access the same variable in memory - at least one of which is a write - that are not ordered by hb.
- Going back to the original program:

Thread 1	*Thread 2*
L1 : R1 = X	L3 : R2 = Y
L2 : Y = R1	L4 : X = R2

- In the execution (L1, L2, L3, L4), there are dataraces between L1 and L3 as well as L2 and L4
- A program is *datarace-free*, if no sequentially consistent execution contains a datarace.

Sequentially Consistent Dataraces

- Consider the program (X and Y are initially 0):

Thread 1	*Thread 2*
`L1: if X == 1`	`L3: if Y == 1`
`L2:      Y = 1`	`L4:      X = 1`

- In a sequential execution, there are not dataraces.
- Here is a consistent execution that contains a datrace:

```
L1: read (1, L4)
L2: write 1
                    L3: read (1, L2)
                    L4: write 1
```

- Any Java Memory Model needs to disallow this execution.

Complicated Reordering (*Optional*)

- Consider the program:

Thread 1	*Thread 2*
L1: R1 = X	L5: R3 = Y
L2: R2 = X	L6: X = R3
L3: if R1 == R2	
L4: Y = 2	

- Consider the reordering:

Thread 1	*Thread 2*
L4: Y = 2	L5: R3 = Y
L1: R1 = X	L6: X = R3
L2: R2 = X	

- Then (L4, L1, L2, L5, L6) becomes a valid reordering.

Lack of Total Order (*Optional*)

Thread 1	*Thread 2*	*Thread 3*
L1 : X = 1	L2 : I = X	L4 : I = X
	L3 : I = X	L5 : I = X