Language-Level Memory Models
A Bit of History ...

Java Memory Model is broken [1].

Double-checked locking (DCL) is broken in Java [2]!

Compiler optimization break things [4].

We need a memory model for C++ [6].

DCL is not portable in C++ [3].

Here is a new JMM [5]!

New shiny C++ memory model

Bacon et. al

Manson et. al

Meyers & Alexandrescu

Pugh

Boehm

Boehm & Adve
The Need for a Language-Level Memory Model

• High-level languages did not provide adequate support for writing portable and safe multithreaded code.

• Traditional compiler optimizations for serial code breaks multithreaded code.

• A memory model also informs the programmer how threads may interact through shared memory.

• A memory model informs the programmer what transformation a compiler can / cannot do.
Compiler Optimization that Breaks Multithreaded Code

Ex: Redundant read elimination

Initially, p.x = 0 and p and q may be aliased.

Thread 1:

```c
int i = p.x;
int j = q.x;
int k = p.x;
```

Thread 2:

```c
// p == q
...
p.x = 42;
...
```

Redundant read elimination:

Thread 1's reads appear out of order.
Compiler Optimization that Breaks Multithreaded Code

Ex: Hoisting and register promotion

```
for(p=x; p!=0; p=p->next) {
    if(p->data < 0) { count++; }
}
```

```
r1 = count; // r1 is a register
for(p=x; p!=0; p=p->next) {
    if(p->data < 0) { r1++; }
}
count = r1;
```

If another thread update **count** at the same time, the update will be lost.
Compiler Transformation that Breaks Multithreaded Code

Ex: Updates to fields in struct

Thread 1:
```
x.a = 1;
struct tmp = x;
tmp.a = 1;
x = tmp;
```

Thread 2:
```
x.b = 42;
```

Thread 2's update can be lost.
Instruction reordering causes thread 2 to read uninitialized data.
In C and C++, we can get around reordering with appropriate hardware fences, but this solution is not portable (and there are no user-level fences in Java).
Lazy Initialization of Singleton
(Single-Threaded Version)

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }
    //...
}

Not thread safe; multiple helpers can get created.
Lazy Initialization of Singleton (Thread-Safe Version)

class Foo {
    private Helper helper = null;
    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
            return helper;
        }
        return helper;
    }
    ...
}

The code performs synchronization every time getHelper() is called.
Can we avoid synchronization after the helper is allocated?
Broken Double-Checked Locking Idiom

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
    //...
}
Broken Double-Checked Locking Idiom

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
    //...
}
Broken Double-Checked Locking Idiom

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
    //...
}

Instructions within a synchronized block can be reordered!

Another thread can see the helper pointer being initialized without the object construction completes.
Broken Double-Checked Locking Idiom

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                Helper h = helper;
                if (h == null)
                    synchronized (this) {
                        h = new Helper();
                    } // force a fence
                helper = h;
            }
        }
        return helper;
    }
    //...
}
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                Helper h = helper;
                if (h == null)
                    synchronized (this) {
                        h = new Helper();
                        } // force a fence
                helper = h;
            }
        }
        return helper;
    }
    //...
}
Broken Double-Checked Locking Idiom

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                Helper h = helper;
                if (h == null)
                    synchronized (this) {
                        h = new Helper();
                    } // force a fence
                helper = h;
            }
        }
        return helper;
    }
    //...
}
Broken Double-Checked Locking Idiom

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                Helper h = helper;
                if (h == null)
                    synchronized (this) {
                        h = new Helper();
                        helper = h;
                    }
            }
        }
        return helper;
    }
    //...
}
The New Java Memory Model

• Goal #1: weak enough to allow as much standard compiler optimizations as possible.
• Goal #2: simple enough to be usable.
• Goal #3: strong enough to preclude behaviors that can pose security hazard (e.g. out-of-thin air values), even for programs with data races.
• A data-race-free (DRF) model:
  – if the code is data-race free, the execution must be sequentially consistent;
  – otherwise, some safety guarantee, but no longer SC.
• Officially incorporated into Java 5.
What Is Data-Race Freedom? (Definition I)

• **Memory location**: each scalar value occupies a separate memory location (1/2/4/8 bytes).

• Types of actions:
  – **data actions**: regular load (read) and store (write)
  – **synchronization actions**: monitor enter (lock) and monitor exit (unlock)
    volatile load, volatile store,
    thread create and the first action of the created thread,
    thread join and the last action of the terminating thread.
What Is Data-Race Freedom? (Definition II)

- Intra-thread execution: actions within a thread are related by its program order
- Inter-thread execution: actions between threads are related by synchronizes-with edges (s.w.):
  - unlock action s.w. subsequent lock action of the same lock
  - a write to volatile var w.s. all subsequent reads to the same volatile var.
  - thread create s.w. first action of the new thread
  - last action of a terminating thread s.w. thread join
  - default initialization of any variable s.w. first action of any thread
- Happens-Before relation: transitive closure of the program order + order imposed by synchronizes-with edges
What Is Data-Race Freedom? (Definition III)

• An execution contains a data race if it contains two memory actions from different threads that are:
  – not ordered by happens-before
  – conflicting (at least one of them is a write)
  – at least one of them is a data operation (i.e., non volatile access)

• A program is data-race free if no sequentially consistent execution contains a data races.
NOT A Data-Race-Free Program

Initially, $x = 0$ and $y = 0$

Thread 1:

```plaintext
r1 = x;
y = 42;
```

Thread 2:

```plaintext
r2 = y;
x = 42;
```

Data races on $x$ and $y$, so the execution can possibly result $r1 = 42$ and $r2 = 42$. 
A Data-Race-Free Program

Initially, $x = 0$ and $y = 0$

Thread 1:

```
r1 = x;
if (r1 != 0)
y = 42;
```

Thread 2:

```
r2 = y;
if (r2 != 0)
x = 42;
```

Any sequentially consistent execution does not cause the write of $x$ and $y$ to occur, so there is no data race. So we should never see $r1 = 42$ and $r2 = 42$. 
The New Java Memory Model

• Goal #1: weak enough to allow as much standard compiler optimizations as possible.
• Goal #2: simple enough to be usable.
• Goal #3: strong enough to preclude behaviors that can pose security hazard (e.g. out-of-thin air values), even for programs with data races.

• A data-race-free (DRF) model:
  – if the code is data-race free, the execution must be sequentially consistent;
  – otherwise, some safety guarantee, but no longer SC.
• Officially incorporated into Java 5.

Much of the complication in JMM stems from Goal #3.
How to Prevent Compiler Optimization from Breaking Your Code

Initially, \( p.x = 0 \) and \( p \) and \( q \) may be aliased.

Thread 1:

\[
\begin{align*}
\text{int } i &= p.x; \\
\text{int } j &= q.x; \\
\text{int } k &= p.x;
\end{align*}
\]

Thread 2:

\[
\begin{align*}
\ldots \\
p.x &= 42; \\
\ldots \\
\end{align*}
\]

Redundant read elimination:

Declare \( p.x \) to be volatile!
How to Prevent Compiler Optimization from Breaking Your Code

```java
for(p=x; p!=0; p=p->next) {
    if(p->data < 0) { count++;
}
}
```

hoisting and register promotion:

```java
r1 = count; //r1 is a register
for(p=x; p!=0; p=p->next) {
    if(p->data < 0) { r1++;
}
}
count = r1;
```

Problem: lost update if another thread update `count` at the same time.

Use `AtomicInteger` if multiple threads are updating `count`. 
How to Prevent Compiler Optimization from Breaking Your Code

Thread 1:

```c
struct tmp = x;
tmp.a = 1;
x = tmp;
```

Thread 2:

```c
x.b = 42;
```

Compilers don't do that unless the fields are bit fields.
How to Prevent Compiler Optimization from Breaking Your Code

Thread 1:
- `data = ...`
- `flag = true;`

Thread 2:
- `while(!flag){}`
- `res = data;`
- `while(!flag) {}`

Prevent reordering by declaring `flag` to be `volatile`. 
Double-Checked Locking that Works

class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
    //...
}

//...
tmp = ...
tmp.field1 = ...;
tmp.field2 = ...;
helper = tmp;
Double-Checked Locking that Works

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

```java
//...
tmp = ...
tmp.field1 = ...;
tmp.field2 = ...;
helper = tmp;
```

A volatile read cannot be reordered with following instructions, and the volatile write cannot be reordered with previous instructions.
So Are We Done?

Java Memory Model is broken [1].

Double-checked locking (DCL) is broken in Java [2]!

Compiler optimization break things [4].

We need a memory model for C++ [6].

DCL is not portable in C++ [3].

New shiny C++ memory model

Here is a new JMM [5]!

JMM is still broken [7].
The New Java Memory Model

• Goal #1: weak enough to allow as much standard compiler optimizations as possible.
• Goal #2: simple enough to be usable.
• Goal #3: strong enough to preclude behaviors that can pose security hazard (e.g. out-of-thin air values), even for programs with data races.

• A *data-race-free (DRF) model*:
  – if the code is *data-race free*, the execution must be sequentially consistent;
  – otherwise, some safety guarantee, but no longer SC.
• Officially incorporated into Java 5.

If your program contains data races, the JMM may not accurately predict the program behaviors.
The C++ Memory Model

• Goal #1: weak enough to allow as much standard compiler optimizations as possible.
• Goal #2: simple enough to be usable.
• Goal #3: strong enough to preclude behaviors that can pose security hazard (e.g. out of thin air values), even for programs with data races.

• A **data-race-free (DRF) model**:
  – if the code is **data-race free**, the execution must be sequentially consistent;
  – otherwise, some safety guarantee, but no longer SC.

Do NOT write programs with data races!
Data Race Can Bite

Initially, $a = 0$ and $b = 1$

Thread 1:
\[
\begin{align*}
    r1 &= a; \\
    r2 &= a; \\
    \text{if} \ (r1 == r2) \\
    b &= 2;
\end{align*}
\]

Thread 2:
\[
\begin{align*}
    r3 &= b; \\
    a &= r3;
\end{align*}
\]

Is $r1 == r2 == r3 == 2$ possible?

Not in any SC execution.
Data Race Can Bite

Initially, a = 0 and b = 1

Thread 1:

```java
r1 = a;
r2 = a;
if (r1==r2)
b = 2;
```

Thread 2:

```java
r3 = b;
a = r3;
```

Compiler transformation:

```
r1 = a;
r2 = r1;
if (r1==r2)
b = 2;
```
Data Race Can Bite

Initially, a = 0 and b = 1

Thread 1:

```java
r1 = a;
r2 = a;
if (r1==r2)
    b = 2;
```

Thread 2:

```java
r3 = b;
a = r3;
```

Now it's possible to get
r1 == r2 == r3 == 2!

Claim: the reordering would not have been visible to the programmer if it weren't for the races on a and b.
Or, declare them to be volatile.
Initially, `count` is 17.

Thread 1:
```
count = 0;
f(positive);
```

Thread 2:
```
f(positive);
count = 0;
```

The variable `count` is only ever written with 0, because everything in the list is positive.

Both threads are unconditionally writing 0 to it, so the race is benign, right?
Benign Race Can Bite

What could possibly go wrong?

Both threads are unconditionally writing 0 to it, so the race is benign, right?
Benign Race Can Bite

Initially, \texttt{count} is 17.

Thread 1:

```c
count = 0;
f(positive);
```

Thread 2:

```c
f(positive);
count = 0;
```

The write to \texttt{count} gives the compiler the license to hoist and do register promotion.

```c
f(p) {
    r1 = count; //r1 is a register
    for(p=x; p!=0; p=p->next) {
        if(p->data < 0) { r1++; }
    }
    count = r1;
}
```
Benign Race Can Bite

Initially, \texttt{count} is 17.

Thread 1:
\begin{verbatim}
count = 0;
rl = count;
count = r1;
\end{verbatim}

Thread 2:
\begin{verbatim}
r2 = count;
count = r2;
count = 0;
\end{verbatim}

Interleaving that gets you in trouble:
\begin{verbatim}
count = 0;
r2 = count;  //17
count = 0;
count = r2;  //17
rl = count;  //17
count = r1;  //17
\end{verbatim}

Final value of \texttt{count} is 17, as if write never occurred!
Summary

The C++ Memory Model
• racy programs have undefined behaviors
• focus on performance
• atomic<T> with
  memory_order_seq_cst
  memory_order_acq_rel
  memory_order_acquire
  memory_order_release
  memory_order_consume
  memory_order_relaxed

The Java Memory Model
• tried to define racy programs' behaviors
• focus on security
• volatile variables for synchronization idiom

Like volatile in Java

The others have looser ordering constraints

Do NOT write programs with data races!
References