An Operational Semantics including volatile for Safe Concurrency

John Boyland
University of Wisconsin-Milwaukee

FTfJP ‘08
Summary

• Volatility is useful and non-trivial;
• Previous semantics used for type systems omitted volatile;
• We use “write keys” to indicate which locations a thread can legally access;
• We can model JMM-inspired “correct synchronization.”
Unsafe Compound

- Abstract mutable interface:

```java
interface CompoundData {
    public void mutate();
    public int compute();
}
```

- Not safe (in general) to interleave
Race Conditions

• Example

```c
int race(CompoundData d) {
    fork { d.mutate(); }
    return d.compute();
}
```
Race Conditions

• Example

```c
int race(CompoundData d) {
    fork { d.mutate(); }
    return d.compute();
}
```
Race Conditions

• Example

```c
int race(CompoundData d) {
    fork {
        d.mutate();
    }
    return d.compute();
}
```

```c
readfield 0xffac4,f
writefield 0xfffac4,f = ..
```
What’s the problem?

Of course, it’s a semantic race, but worse
1. access while invariants invalidated;
2. sequential consistency not guaranteed!
   • some writes may be observed;
   • others not, even if earlier.
Non-solutions

1. Hope problem “never” happens;

2. Make all fields volatile everywhere:
   - invariants weakened;
   - optimization all but impossible.
Safe Compound (synch)

class Traditional {
    private CompoundData base;
    public void mutate() {
        synchronized (this) {
            base.mutate();
        }
    }
    public int compute() {
        synchronized (this) {
            return base.compute();
        }
    }
}
Safe Compound (synch)

class Traditional {
    private CompoundData base;
    public void mutate() {
        synchronized (this) {
            base.mutate();
        }
    }
    public int compute() {
        synchronized (this) {
            return base.compute();
        }
    }
}
synch Advantages

+ Race conditions avoided:
  • broken invariants protected;
  • sequential consistency restored.

- Execution overhead of locks;
- Danger of deadlock.

But if mutation is rare, we can use an interesting design pattern with volatile ...
class UsingVolatile {
    private volatile CompoundData base;
    public void mutate() {
        synchronized (this) {
            base = base.clone().mutate();
        }
    }
    public int compute() {
        return base.compute();
    }
}
class UsingVolatile {
    private volatile CompoundData base;
    public void mutate() {
        synchronized (this) {
            base = base.clone().mutate();
        }
    }
    public int compute() {
        return base.compute();
    }
}
How to Prove Safety?

Previous Way:

1. Define semantics;
2. Define type system;
3. Prove subject reduction (soundness);
4. Prove that type system avoids races.
How to Prove Safety?

Previous Way:
1. Define semantics;
2. Define type system;
3. Prove subject reduction (soundness);
4. Prove that type system avoids races.

Current semantics omit volatile
How to Prove Safety?

Previous Way:
1. Define semantics;
2. Define type system;
3. Prove subject reduction (soundness);
4. Prove that type system avoids races.

Current semantics omit volatile

Complex proof using global reasoning
How to Prove Safety!

New way:

1. Define semantics; (DONE)
2. Define type system;
3. Prove subject reduction (soundness);
4. Prove that type system avoids races.
How to Prove Safety!

New way:

1. Define semantics; (DONE)
2. Define type system;
3. Prove subject reduction (soundness);
4. Prove that type system avoids races.

Simple semantics: no thread interleaving
A program is *correctly synchronized* if and only if in all sequentially consistent executions, all conflicting accesses [RW, WR, WW] to non-volatile variables are ordered by “happens-before” edges. [JMM = Java Memory Model]

- Only correctly synchronized programs can rely on sequential consistency.
“Happens Before”

- Intra-thread program order PLUS “synchronizes with” edges:
  1. `fork` to first instruction in thread;
  2. last instruction in thread to `join`;
  3. release lock to acquire lock;
  4. volatile write to volatile read.
“Happens Before”

- Intra-thread program order PLUS “synchronizes with” edges:
  1. `fork` to first instruction in thread;
  2. last instruction in thread to `join`;
  3. release lock to acquire lock;
  4. volatile write to volatile read.

Volatile cannot be ignored!
Example

- fork
- writev
- synch
- join
- writev
- readv
- synch
- join
Example

fork

writev

join

writev

synch

readv

synch

join
Example

fork
writev
join

writev
synch

readv
synch

join
Example

- fork
- join
- writev
- synch
- readv
- synch
- synch
- synch
- writev
- join
Example

write
fork
writev
join

writev
synch
readv
synch
synch
writev
join
Example
Example

fork
write
writev
join

read
readv

writev

synch
synch
synch
join
Example

Synchronization Error!

write
writev
read
synch
fork
ready
join
join
New Semantics

1. Start with a conventional store semantics;

2. Add concept of “write keys”:
   - Every thread knows some keys (knowledge never lost);
   - New keys generated at writes;
   - Keys transferred through memory;

3. Knowledge required for access.
Simulate “happens before”

1. `fork` passes keys to new thread;
2. `join` picks up keys from thread;
3. `release` stores keys in mutex, `acquire` picks up keys from mutex;
4. `volatile` write adds keys to field, `volatile` read picks up keys from field.
Write Keys

fork

writev

synch

writev

join

synch

readv

writev

join
Write Keys

- fork
- join
- writev
- readv
- synch
- synch
- synch
- synch
- join
- writev
Write Keys

- write
- fork
- join
- writev
- synch
- synch
- synch
- ready
- writev
- synch
- synch
- synch
- join
- join
Write Keys

- write
- fork
- join
- writev
- synch
- writev
- ready
- synch
- join
Write Keys

- write
- fork
- join
- writev
- writev
- synth
- readyv
- synth
- synth
- join
Write Keys

- write
- fork
- join
- writev
- synch
- writev
- ready
- synch
- join
Write Keys

- write
- fork
- join
- writev
- synch
- write
- join
- writev
- synch
- ready
- join
\[ \mu(o.f) = (\{w\}, \_ ) \]

\[ w \in \kappa(p) \quad f \not\in F_v \quad w' \text{ arbitrary} \]

\[ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad \kappa' = \kappa[p \mapsto \{w'\}] \]

\[ (\mu; \theta; \kappa; o. f := o') \xrightarrow{p} (\mu'; \theta; \kappa'; o') \]
Thread $p$ performs a write.

$\mu(o.f) = (\{w\}, -)$

$w \in \kappa(p)$ \quad $f \notin F_V$ \quad $w'$ arbitrary

$\mu' = \mu[\ o.f \mapsto (\{w'\}, o')\ ]$ \quad $\kappa' = \kappa[\ p \mapsto \{w'\}\ ]$

$(\mu; \theta; \kappa; o. f := o') \xrightarrow{p} (\mu'; \theta; \kappa'; o')$

$g$
Thread $p$ performs a write.

\[ \mu(o.f) = (\{w\}, -) \]
\[ w \in \kappa(p) \quad f \notin F_V \quad w' \text{ arbitrary} \]
\[ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad \kappa' = \kappa[p \mapsto \{w'\}] \]

\[ (\mu; \theta; \kappa; \text{Field Store} \quad \boxed{o.f := o'}) \xrightarrow{p} (\mu'; \theta; \kappa'; \boxed{o'}) \]

Field Store “memory”
E-Write

\[ \mu(o.f) = (\{w\}, -) \]

\[ w \in \kappa(p) \quad f \not\in F_V \quad w' \text{ arbitrary} \]

\[ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad \kappa' = \kappa[p \mapsto \{w'\}] \]

\[ (\mu; \theta; \kappa; o.f := o') \xrightarrow{p} g(\mu'; \theta; \kappa'; o') \]

Thread \( p \) performs a write.

Field Store “memory”

Known write keys
Thread $p$ performs a write.

Field Store “memory”

Known write keys

\[
\begin{align*}
\mu(o.f) &= (\{w\}, -) \\
w &\in \kappa(p) \\
f &\not\in F_V \\
\mu' &= \mu[o.f \mapsto (\{w'\}, o')] \\
w' &\text{ arbitrary} \\
\kappa' &= \kappa[p \mapsto \{w'\}] \\
(\mu; \theta; \kappa; o.f := o') &\xrightarrow{p} (\mu'; \theta; \kappa'; o') \\
\xrightarrow{g}
\end{align*}
\]

E-Write

\[\mu(o.f) = (\{w\}, -)\]

\[w \in \kappa(p)\]

\[f \not\in F_V\]

\[w' \text{ arbitrary}\]

\[\mu' = \mu[o.f \mapsto (\{w'\}, o')]\]

\[\kappa' = \kappa[p \mapsto \{w'\}]\]

\[(\mu; \theta; \kappa; o.f := o') \xrightarrow{p} (\mu'; \theta; \kappa'; o')\]

\[(\mu'; \theta; \kappa'; o') \xrightarrow{g}\]
Field Store "memory"

Field’s current write key is $w$.

**E-Write**

\[ \mu(o.f) = (\{w\}, -) \]

$w \in \kappa(p)$ \hspace{1cm} $f \notin F_V$ \hspace{1cm} $w'$ arbitrary

\[ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \hspace{1cm} \kappa' = \kappa[p \mapsto \{w'\}] \]

\[ (\mu; \theta; \kappa; o.f := o') \xrightarrow{p} (\mu'; \theta; \kappa'; o') \]

Known write keys
E-Write

\[ \mu(o.f) = (\{w\}, -) \]

\[ w \in \kappa(p) \]

\[ f \notin F_v \]

\[ w' \text{ arbitrary} \]

\[ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \]

\[ \kappa' = \kappa[p \mapsto \{w'\}] \]

\[ (\mu; \theta; \kappa; o.f := o') \xrightarrow{p} (\mu'; \theta; \kappa'; o') \]

Field’s current write key is \(w\).

(which which thread \(p\) knows)
E-Write

\[
\mu(o.f) = (\{w\}, -)
\]

\[
w \in \kappa(p) \quad f \notin F_V
\]

\[
w' \text{ arbitrary}
\]

\[
\mu' = \mu[o.f \mapsto (\{w'\}, o')]
\]

\[
\kappa' = \kappa[p \mapsto \{w'\}]
\]

\[
(\mu; \theta; \kappa; o.f := o') \xrightarrow{p} (\mu'; \theta; \kappa'; o')
\]

\[
\xrightarrow{g}
\]

Memory updated with new write key and value.
E-Write

\[ \mu(o.f) = (\{w\}, \_ \) \]

\[ w \in \kappa(p) \]

\[ f \not\in F_V \]

\[ w' \text{ arbitrary} \]

\[ \kappa' = \kappa[p \mapsto \{w'\}] \]

\[ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \]

\[ (\mu; \theta; \kappa; o . f := o') \xrightarrow{p} (\mu'; \theta; \kappa'; o') \]

Memory updated with new write key and value. (which may be one no thread knows)
E-Write

\[ \mu(o.f) = (\{w\}, -) \]

\[ w \in \kappa(p) \quad f \notin F_V \]

\[ w' \text{ arbitrary} \]

\[ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \]

\[ \kappa' = \kappa[p \mapsto \{w'\}] \]

\[ (\mu; \theta; \kappa; o.f := o') \xrightarrow{p} (\mu'; \theta; \kappa'; o') \]

\[ g \]

Memory updated with new write key and value.

Thread \( p \) now knows the new key.
Write-Key Errors

- A thread is ready to access a field (either a read or a write);
- The write key for this field is some $w$;
- The thread does not know $w$;
- The thread blocks.
Theorem

The following three statements about a program are equivalent:

1. The program never has a write key error;
2. The program is correctly synchronized;
3. The program has no race conditions.

(Proved in Twelf.)
What is missing

- No guarantee that race conditions will be detected (in a particular run);
- No JMM-compliant semantics of incorrectly synchronized programs;
- No \texttt{wait}; no primitives; no dynamic dispatch; ...
- No type system.
Conclusions

1. Volatile variables are useful and non-trivial;

2. Write keys capture essence of “happens before” relation without thread communication for volatile / mutex;

3. Race free = correctly synchronized.