THE JAVA MEMORY MODEL

J. Manson, W. Pugh, and S. Adve

Brian Gernhardt - Feb 21, 2013
Parallel and Distributed Systems

INTRODUCTION

MEMORY MODEL

Limits on order of memory accesses

Programmer	Compiler & Hardware
Ease of Use	**Optimizations**

SEQUENTIAL CONSISTENCY

```
x = y = 0
```

- Accesses in program order
- Easy to use
- No reordering
- Few optimizations

```
r2 = x      r1 = y
y = 1       x = 2
```

r1	0	1	0	1
r2	0	0	2	2

CORRECTLY SYNCHRONIZED

- No data races in *any* sequentially consistent execution
- Example always has write next to read
  - Always has a race

```
r2 = x      r1 = y
y = 1       x = 2
```

DATA RACE FREE

- If you mark synchronization
  - **`volatile`** and **`synchronized`**
- Then you get Sequentially Consistent behavior

```
volatile int x, y;

synchronized {  synchronized {
  r2 = x          r1 = y
  y = 1           x = 2
}               }
```

DEFINITIONS

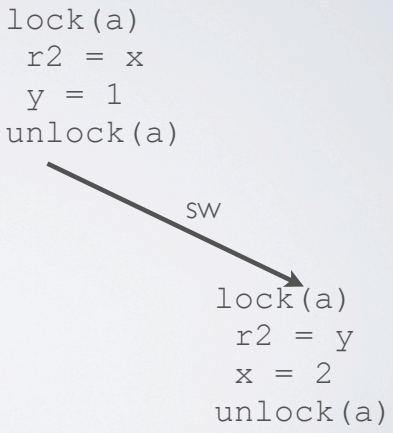
SYNCHRONIZATION ACTION

- Lock
- Unlock
- Read volatile
- Write volatile

```
lock(a)     lock(a)
 r2 = x      r2 = y
 y = 1       x = 2
unlock(a)   unlock(a)
```

SYNCHRONIZES WITH

- Order synchronization actions
- Unlocks before locks
- Volatile writes before reads
- Between threads

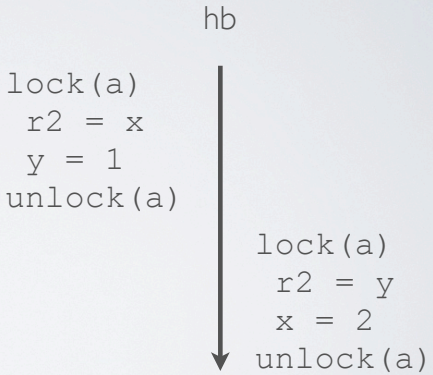


PROGRAM ORDER

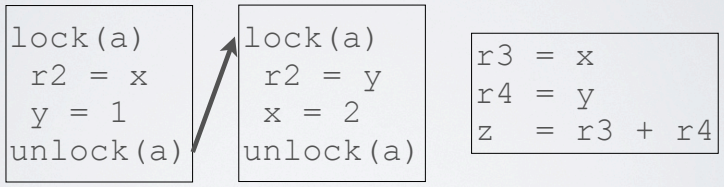
- Order of accesses in code
- Only within same thread

HAPPENS-BEFORE

- Synchronizes-with combined with program order
- Transitive Closure
  - If a → b → c
  - Then a → c



HAPPENS-BEFORE



CONFLICTING ACCESS

- Two or more accesses to same variable
- At least one is a write

```
r1 = x
r2 = x
x = r3
```

DATA RACE

- Different threads
- Conflicting access
- No happens-before order

```
r2 = x      r1 = y
y = 1       x = 2
```

HAPPENS-BEFORE MODEL

HAPPENS-BEFORE MODEL

HAPPENS-BEFORE MODEL

- Each thread acts as if it was alone
  - Given that it reads the same values
- Non-volatile reads must be happens-before consistent
- Volatile reads must be synchronization order consistent

HAPPENS-BEFORE CONSISTENCY

- Determines when *r* can read the value from *w*
  - $r \rightarrow w$ *r* can't see *w*
  - $w \rightarrow w' \rightarrow r$ If *r* sees *w*, then *w'* doesn't exist

SYNCHRONIZATION ORDER CONSISTENCY

- Synchronization order is consistent with program order
  - Can't reorder unlocks before locks, etc.
- Volatile read sees the previous write in synchronization order

```
x = 0
volatile ready = false
```

`r1 = x` never happens
or
`r1 == x == 1`

CAUSALITY

UNDEFINED BEHAVIOR

- Means anything can happen
- Hard to understand

```
x = y = 0

r2 = x      r1 = y
y = 1       x = 2

r1 = r2 = 42
```

- Bad for security
  - 42 isn't bad
  - Password is

CORRECTLY SYNCHRONIZED

```
r1 = x
if (r1 ≠ 0)
   y = 42
```

```
r2 = y
if (r2 ≠ 0)
   x = 42
```

- Sequentially consistent
  - Reads r1 before setting y
  - Reads r2 before setting x
- No data race

CIRCULAR LOGIC

```
r1 = x
if (r1 ≠ 0)
   y = 42
```

```
r2 = y
if (r2 ≠ 0)
   x = 42
```

- No happens before order between threads
- Means reads are allowed to see writes
- Because reads see writes, writes can occur
- Don't want this

CIRCULAR LOGIC?

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

```
r3 = b
a  = r3
```

Can `r1 == r2 == r3 == 2` ?

OPTIMIZATIONS

Duplicate Read

```
r1 = a                    r3 = b
r2 = a                    a  = r3
if (r1 == r2)
  b = 2
```

OPTIMIZATIONS

```
r1 = a                    r3 = b
r2 = r1                   a  = r3
if (r1 == r2)
  b = 2
```

Always true

OPTIMIZATIONS

```
r1 = a                    r3 = b
r2 = r1                   a  = r3
if (true);
  b = 2
```

OPTIMIZATIONS

```
b = 2
r1 = a                    r3 = b
r2 = r1                   a  = r3
if (true);
```

Now `r1 == r2 == r3 == 2`

WHAT'S THE DIFFERENCE?

```
r1 = x                    r1 = a
if (r1 ≠ 0)               r2 = a
   y = 42                 if (r1 == r2)
                            b = 2
```

```
r2 = y                    r3 = b
if (r2 ≠ 0)               a  = r3
   x = 42
```

With Sequential Consistency:

Writes never happen Write always happens

JAVA MEMORY MODEL

WELL-BEHAVED EXECUTION

- Justifies reordering by iteratively building execution
  - Early execution allowed if not caused by data race
- Maintain consistencies from happens-before model
  - Sequential consistency for proper synchronization
- Well-behaved for races

COMMITTING ACTIONS

- Can commit any reads or writes
- Reads can only see values from committed writes
- Before committing, reads only see happens-before writes
  - Use these values to justify committing more actions
- After committing, read can see any write

JUSTIFYING EARLY WRITE

```
  a = b = 0

r1 = a
r2 = a
if (r1 == r2)
  b = 2

r3 = b
a  = r3
```

- Start with no committed actions

JUSTIFYING EARLY WRITE

```
  a = b = 0

r1 = a
r2 = a
if (r1 == r2)
  b = 2

r3 = b
a  = r3
```

- Can commit initialization because it happens before everything else
- Can commit all the reads and write of a

JUSTIFYING EARLY WRITE

```
  a = b = 0

0 r1 = a
0 r2 = a
  if (r1 == r2)
    b = 2

0 r3 = b
  a  = r3
```

- Must assume reads see the happens-before writes (initialization)

JUSTIFYING EARLY WRITE

```
    a = b = 0

  0 r1 = a
  0 r2 = a
true if (r1 == r2)
      b = 2

  0 r3 = b
    a  = r3
```

- These reads mean that if condition is true

JUSTIFYING EARLY WRITE

```
      a = b = 0

   0 r1 = a
   0 r2 = a
true if (r1 == r2)
       b = 2

   0 r3 = b
     a  = r3
```

- We can now also commit writing b

COMMIT SEES RACE

```
      a = b = 0

     r1 = a
     r2 = a
     if (r1 == r2)
       b = 2

   2 r3 = b
     a  = r3
```

- b = 2 committed
- So r3 = b can see 2

COMMIT SEES RACE

```
      a = b = 0

     r1 = a
     r2 = a
     if (r1 == r2)
       b = 2

   2 r3 = b
   2 a  = r3
```

- Write of a uses value seen by read into r3

COMMIT SEES RACE

```
      a = b = 0

     r1 = a
     r2 = a
     if (r1 == r2)
       b = 2

     r3 = b
     a  = r3
```

- Committed reads can see writes committed at same time.

COMMIT SEES RACE

```
    a = b = 0

2 r1 = a
2 r2 = a
  if (r1 == r2)
    b = 2

2 r3 = b
2 a  = r3
```

- Other reads can see committed write of a

COMMIT SEES RACE

```
    a = b = 0

2 r1 = a
2 r2 = a
  if (r1 == r2)
    b = 2

2 r3 = b
2 a  = r3
```

- `r1 == r2 == r3 == 2`

IMPLEMENTERS

- Weak control dependence
  - Compiler cannot assume loops will end.
  - Can not move statements ahead of loops
- Most reordering is allowed, within limits
- Can ignore thread-local synchronization

PROGRAMMERS

- Correctly synchronized code acts sequentially consistent
- JVM does not guarantee pre-emption or fairness
  - Due to complications with priorities and real-time
- Java Memory Model created via community discussion
  - Some fine details are mostly matters of taste