Concurrency WS 2010/2011
The Java Memory Model

Peter Thiemann

November 2, 2010

Outline

Java Memory Model

- Java does **not** guarantee linearizability or sequential consistency

Java Memory Model

- Java does **not** guarantee linearizability or sequential consistency
- Sequential consistency inhibits widely used compiler optimizations that reorder memory reads and writes
  - register allocation
  - common subexpression elimination
  - redundant read elimination

Relaxed Memory Model

Fundamental Property If a program's sequentially consistent executions follow certain rules, then every execution of that program in the relaxed model will still be sequentially consistent.

Reading and Writing

- Objects reside in shared memory
- Each thread has a local cached copy of fields it has read or written
- A write to a field may not propagate to shared memory
- A read of a field may see a cached value instead of the one in shared memory
- Own, local reads and writes happen in order

Anti-Pattern: Double-Checked Locking

```
1   private Singleton instance = null;
2   public static Singleton getInstance() {
3     if (instance == null) {
4       synchronized (Singleton.class) {
5         if (instance == null)
6           instance = new Singleton ();
7       }
8     }
9     return instance;
10  }
```

Double-Checked Locking is Incorrect!

- The assignment to `instance` in line 6 may be written to memory before the constructor is finished initializing the object.
- Another thread's invocation of `getInstance` would find that `instance != null` in line 3 and return the reference to the unfinished object.

Synchronization Events

- Certain statements are synchronization events
- Not necessarily atomicity or mutual exclusion
- Reconciliation of local cache with shared memory
  - flushing local writes
  - invalidating cached reads
- Synchronization events are linearizable: totally ordered and all threads agree

Locks and Synchronized Blocks

- Mutual exclusion by
  - entering a **`synchronized`** block or method
  - acquiring an explicit lock
- If all accesses to a field protected by the same lock, then reads-writes to that field are linearizable:
  - `unlock()` writes back changed fields to shared memory
  - `lock()` invalidates the cache, forcing a reread from shared memory

Volatile Fields

- Fields declared `volatile` are linearizable
- Reading from volatile is like acquiring a lock
  invalidates the cache, read from shared memory
- Writing to volatile is like releasing a lock
  writes through to shared memory

Volatile Fields

- Fields declared `volatile` are linearizable
- Reading from volatile is like acquiring a lock
  invalidates the cache, read from shared memory
- Writing to volatile is like releasing a lock
  writes through to shared memory

Attention

- Multiple read-writes are not atomic
- Typical usage pattern: single writer / multiple readers

Volatile Fields

- Fields declared **`volatile`** are linearizable
- Reading from volatile is like acquiring a lock
  invalidates the cache, read from shared memory
- Writing to volatile is like releasing a lock
  writes through to shared memory

Attention

- Multiple read-writes are not atomic
- Typical usage pattern: single writer / multiple readers

Linearizable Memory

- `AtomicReference<T>`, `AtomicInteger`
- Methods `compareAndSet()`, `set()` like writes
- Method `get()` like read

Final Fields

- A `final` field cannot be modified once it has been initialized.
- Initialized in the constructor
- Under simple rules, the correct value of a final field will be visible to other threads without synchronization.

Example: Constructor with Final Field

```
1  class FinalFieldExample {
2    final int x; int y;
3    static FinalFieldExample f;
4    public FinalFieldExample() {
5      x = 3;
6      y = 4;
7    }
8    static void writer() {
9      f = new FinalFieldExample ();
10   }
11   static void readers() {
12     if (f != null) {
13       int i = f.x; int j = f.y;
14       // i == 3 is guaranteed
15       // no guarantee about y's value
16     }
17   }
18 }
```

Incorrect EventListener Class

```
1 public class EventListener {
2   final int x;
3   public EventListener (EventSource eventSource) {
4     eventSource.registerListener(this);
5   }
6   public onEvent(Event e) {
7     // handle the event
8   }
9 }
```

- `onEvent` may be invoked
  - after the listener is registered
  - but before the constructor is completed, i.e., before the value of `x` is flushed to shared memory

Outline

Example Programs

- Example programs taken from “Java Concurrency in Practice” by Brian Goetz and others
- source available from `http://www.javaconcurrencyinpractice.com/`

Thread Safety

Stateless objects are always thread-safe

```
15 @ThreadSafe
16 public class StatelessFactorizer extends GenericServlet implem
17
18     public void service(ServletRequest req, ServletResponse re
19         BigInteger i = extractFromRequest(req);
20         BigInteger[] factors = factor(i);
21         encodeIntoResponse(resp, factors);
22     }
```

Atomicity

Don't do this

```
16 public class UnsafeCountingFactorizer extends GenericServlet i
17     private long count = 0;
18
19     public long getCount() {
20         return count;
21     }
22
23     public void service(ServletRequest req, ServletResponse re
24         BigInteger i = extractFromRequest(req);
25         BigInteger[] factors = factor(i);
26         ++count;
27         encodeIntoResponse(resp, factors);
28     }
```

Lazy Initialization

Don't do this

```
14 public class LazyInitRace {
15     private ExpensiveObject instance = null;
16
17     public ExpensiveObject getInstance() {
18         if (instance == null)
19             instance = new ExpensiveObject();
20         return instance;
21     }
22 }
23
24 class ExpensiveObject { }
```

Safe Lazy Initialization

```
13 public class SafeLazyInitialization {
14     private static Resource resource;
15
16     public synchronized static Resource getInstance() {
17         if (resource == null)
18             resource = new Resource();
19         return resource;
20     }
21
22     static class Resource {
23     }
24 }
```

Eager Initialization

```
13 public class EagerInitialization {
14     private static Resource resource = new Resource();
15
16     public static Resource getResource() {
17         return resource;
18     }
19
20     static class Resource {
21     }
22 }
```

More Lazy Initialization

```
13 public class ResourceFactory {
14     private static class ResourceHolder {
15         public static Resource resource = new Resource();
16     }
17
18     public static Resource getResource() {
19         return ResourceFactory.ResourceHolder.resource;
20     }
21
22     static class Resource {
23     }
24 }
```

One state variable

```
17 public class CountingFactorizer extends GenericServlet impleme
18     private final AtomicLong count = new AtomicLong(0);
19
20     public long getCount() { return count.get(); }
21
22     public void service(ServletRequest req, ServletResponse re
23         BigInteger i = extractFromRequest(req);
24         BigInteger[] factors = factor(i);
25         count.incrementAndGet();
26         encodeIntoResponse(resp, factors);
27     }
```

Locking: More than one state variable

Don't do this

```
18 public class UnsafeCachingFactorizer extends GenericServlet im
19     private final AtomicReference<BigInteger> lastNumber
20             = new AtomicReference<BigInteger>();
21     private final AtomicReference<BigInteger[]> lastFactors
22             = new AtomicReference<BigInteger[]>();
23
24     public void service(ServletRequest req, ServletResponse re
25         BigInteger i = extractFromRequest(req);
26         if (i.equals(lastNumber.get()))
27             encodeIntoResponse(resp, lastFactors.get());
28         else {
29             BigInteger[] factors = factor(i);
30             lastNumber.set(i);
31             lastFactors.set(factors);
32             encodeIntoResponse(resp, factors);
33         }
34     }
```

Correct but inefficient locking

Don't do this

```
17 public class SynchronizedFactorizer extends GenericServlet imp
18     @GuardedBy("this") private BigInteger lastNumber;
19     @GuardedBy("this") private BigInteger[] lastFactors;
20
21     public synchronized void service(ServletRequest req,
22                                      ServletResponse resp) {
23         BigInteger i = extractFromRequest(req);
24         if (i.equals(lastNumber))
25             encodeIntoResponse(resp, lastFactors);
26         else {
27             BigInteger[] factors = factor(i);
28             lastNumber = i;
29             lastFactors = factors;
30             encodeIntoResponse(resp, factors);
31         }
32     }
```

Rentrancy

```
1  public class Widget [
2    public synchronzied void doSomething() {
3      ...
4    }
5  }
6
7  public class LoggingWidget extends Widget {
8    public synchronzied void doSomething() {
9      System.out.println(toString() + ": calling doSomething");
10     super.doSomething();
11   }
12 }
```

Locking: More than one state variable

Working example

```
16 public class CachedFactorizer extends GenericServlet implements Servlet {
17     @GuardedBy("this") private BigInteger lastNumber;
18     @GuardedBy("this") private BigInteger[] lastFactors;
19     @GuardedBy("this") private long hits;
20     @GuardedBy("this") private long cacheHits;
21
22     public synchronized long getHits() {
23         return hits;
24     }
25
26     public synchronized double getCacheHitRatio() {
27         return (double) cacheHits / (double) hits;
28     }
29
30     public void service(ServletRequest req, ServletResponse resp) {
31         BigInteger i = extractFromRequest(req);
32         BigInteger[] factors = null;
33         synchronized (this) {
34             ++hits;
35             if (i.equals(lastNumber)) {
36                 ++cacheHits;
37                 factors = lastFactors.clone();
38             }
39         }
40         if (factors == null) {
41             factors = factor(i);
42             synchronized (this) {
43                 lastNumber = i;
44                 lastFactors = factors.clone();
45             }
46         }
47         encodeIntoResponse(resp, factors);
```

Visibility

Don’t do this

```
11 public class NoVisibility {
12     private static boolean ready;
13     private static int number;
14
15     private static class ReaderThread extends Thread {
16         public void run() {
17             while (!ready)
18                 Thread.yield();
19             System.out.println(number);
20         }
21     }
22
23     public static void main(String[] args) {
24         new ReaderThread().start();
25         number = 42;
26         ready = true;
27     }
28 }
```

Unsafe publication

Don’t do this

```
10 class UnsafeStates {
11     private String[] states = new String[]{
12         "AK", "AL" /*...*/
13     };
14
15     public String[] getStates() {
16         return states;
17     }
18 }
```

Unsafe publication II

Don't do this

```
10 public class ThisEscape {
11     public ThisEscape(EventSource source) {
12         source.registerListener(new EventListener() {
13             public void onEvent(Event e) {
14                 doSomething(e);
15             }
16         });
17     }
18
19     void doSomething(Event e) {
20     }
21
22     interface EventSource {
23         void registerListener(EventListener e);
24     }
25
26     interface EventListener {
27         void onEvent(Event e);
28     }
29
30     interface Event {
31     }
32 }
```

Safe publication

```
10 public class SafeListener {
11     private final EventListener listener;
12
13     private SafeListener() {
14         listener = new EventListener() {
15             public void onEvent(Event e) {
16                 doSomething(e);
17             }
18         };
19     }
20
21     public static SafeListener newInstance(EventSource source) {
22         SafeListener safe = new SafeListener();
23         source.registerListener(safe.listener);
24         return safe;
25     }
26
27     void doSomething(Event e) {
28     }
29
30     interface EventSource {
31         void registerListener(EventListener e);
32     }
33
34     interface EventListener {
35         void onEvent(Event e);
36     }
37
38     interface Event {
39     }
40 }
```