Some JMM Issues for Programmers

Doug Lea and William Pugh

http://gee.cs.oswego.edu/

http://www.cs.umd.edu/~pugh

Based in part on slides from talks at JavaOne 2000
Outline

- Programmer view of memory model
  - synchronized, volatile, final

- Some issues and examples
  - When and why to synchronize
  - Initialization
  - Documentation
Three Aspects of Synchronization

- **Atomicity**
  - Locking to obtain mutual exclusion
  - Atomic read/write granularity

- **Visibility**
  - Ensuring that changes to object fields made in one thread are seen in other threads

- **Ordering**
  - Ensuring that you aren’t surprised by the order in which statements are executed
Quiz Time

\[ x = y = 0 \]

Thread 1

\[ x = 1 \]

\[ j = y \]

Thread 2

\[ y = 1 \]

\[ i = x \]

Can this result in \( i = 0 \) and \( j = 0 \)?
Answer: Yes!

How can $i = 0$ and $j = 0$?
How Can This Happen?

- Compiler can reorder statements
- Or keep values in registers
- Processor can reorder them
- On multiprocessor, values held in caches without synchronizing global memory
- Must use synchronization to enforce visibility and ordering
- As well as mutual exclusion
Programmer view of Synchronization

// block until obtain lock
synchronized(anObject) {
    // get main memory value of field1 and field2
    int x = anObject.field1;
    int y = anObject.field2;
    anObject.field3 = x+y;
    // commit value of field3 to main memory
}

// release lock
moreCode();
When Are Actions Visible to Other Threads?

Thread 1

1. $x = 1$
2. unlock $M$

Thread 2

1. lock $M$
2. $i = x$
Volatile Fields

- Volatile reads/writes go directly to memory
- Sequentially consistent with respect to each other and to monitor enter/exit.
- Can’t be cached in registers or local memory
- In quiz example, if x and y are volatile, it is impossible to see i = 0 and j = 0
- Reads and writes of volatile longs and doubles are atomic
- But many JVMs don’t comply
Using Volatile

class Animator implements Runnable {
    volatile boolean stop = false;
    public void stop() { stop = true; }
    public void run() {
        while (!stop)
            oneStep();
    }
    void oneStep() { /*...*/ }
}

- **stop** must be declared volatile
- Otherwise, compiler could keep in register
Limitations of Volatile

class Future {
   // DO NOT USE!
   private volatile boolean ready = false;
   private Object data = null;
   public Object get() {
      if (!ready) return null;
      return data;
   }
   // only one thread ever calls put
   public void put(Object o) {
      data = o;
      ready = true;
   }
}

- Reading volatile does not "guard" read of non-volatile
Final Fields

- JLS ch17 does not even mention final fields
  - In part because it predated "blank finals"
  - So, currently, `final` has no memory semantics

- Any plausible JMM should guarantee that:
  
  A final field initialized in constructor is always read as having its final value
  
  (... unless programmers do something stupid.)

- CPJ2e says this is true now!

- Details are intrinsically messy since there is no syntax to say an array element is final
  - Impacts java.lang.String
JMM impact on programmers

- Programs with a lot of synchronization run slowly
  - Even if they do not generate threads

- Sophisticated programmers who try to minimize synchronization often get it wrong
  - Some clever idioms are not guaranteed to work
  - A clear JMM is needed to guide use

- Unsophisticated programmers often produce unsafe code that "works"
  - Even "works" on Sparc and Intel MPs, but isn’t correct according to any plausible JMM
**Essential synchronization**
is rarely a bottleneck

- Synch only in places where threads interact
  - Needs careful thought, good documentation
  - But can be more error-prone than blanket synchronization
    - Races harder to debug than liveness failures
    - Longer development time

- Synchronizing thread local objects is useless
  - But in practice accounts for majority of synchronizations
Library classes

- Library code cannot know whether it is used by multiple threads.
- Usually synchronized even when multithreaded use extremely rare
  - java.io Streams, Vector, StringBuffer, ...
- The most frequent cause of useless synch
  - In places where not required in current context, but is required in other contexts.
  - Optimization algorithms to detect and eliminate such locks are still not practical.
Writing safe reusable code

How to efficiently synch code when you don’t know usage contexts

- Provide only large units of granularity?
  - Can enable more efficient internals
  - Can lead to ugly, awkward interfaces

- Impose usage restrictions?
  - Examples: Swing, EJB
  - Compliance generally uncheckable

- Supply safe versions layered over unsafe?
  - Example: java.util synchronizedCollection
  - But best MT code is not always synchronized version of best single-threaded code
Lock granularity example

Create new value and add to Hashtable only if key absent

- Naïve unsynched user code is unsafe:
  ```java
  ID getID(String name) {
    ID x = (ID) (hashtab.get(name));
    if (x == null)
      hashtab.put(name, (x = new ID()));
    return x;
  }
  ```

- Synching user method causes useless synch
- Directly support?
  ```java
  Object putIfAbsent(Object k, Callback c)
  ```
Field Access Methods

- Sometimes, synchronizing access methods is obviously the right thing to do:
  ```java
  account.getTotalBalance()
  ```
- Consequent fine-granularity locking is desirable

- In other cases it is a useless bottleneck
  - And doesn’t preserve intended invariants:
    ```java
    x = point.getX(); y = point.getY();
    ```

- Often, the best decision is just to omit access methods
  - `queue` doesn’t need `getSize()`
Unsynchronized Accessors

(or equivalently, public raw field access)

- Sometimes stale values OK
  - or avoidable by using volatile fields
    
    `thermometer.getTemperature()`

- Almost always dangerous for references
  - Read of ref doesn’t guarantee read of field
  - Doesn’t help to synchronize only `setColor`

```java
private Color color;
void setColor(int rgb) {
    color = new Color(rgb);
}

Color getColor() { return color; }
```
Example: demo code

```java
Thread blinker = null;

public void start() {
    blinker = new Thread(this);
    blinker.start();
}

public void stop() {
    blinker = null;
}

public void run() {
    Thread me = Thread.currentThread();
    while (blinker == me) {
        try {
            Thread.sleep(delay);
        } catch (InterruptedException e) {
            return;
        }
        repaint();
    }
}
```
Initialization

- Rules for finals cover only one case of (conceptually) "write once" fields.

- Most other cases require synch:
  - When no-arg constructors are mandated
  - When deserializing
  - Build-then-release patterns

```java
synchronized Bean makeBean() {
    Bean b = new MyBean();
    b.setAttributeA("finalValue");
    return b;
}
```

- Lazy initialization
Lazy Initialization

Basic version:

class Service {
    Parser parser = null;
    public synchronized void command() {
        if (parser == null) {
            parser = new Parser(...);
            doCommand(parser.parse(...));
        }
        // ...
    }
}
Initialization checks

class ServiceV2 {
    Parser parser = null;
    synchronized Parser getParser() {
        Parser p = parser;
        return (p != null) ? p :
            (parser = new Parser());
    }
    public void command(...) {
        doCommand(getParser().parse(...));
    }
}

Isolating checks reduces lock durations, usually improving MT performance
Single−Check

class ServiceV3 { // DO NOT USE
    Parser parser = null;
    Parser getParser() { // no synch
        Parser p = parser;
        return (p != null) ? p :
            (parser = new Parser());
    }
}

- Possible to see p as non−null, but not see p’s fields initialized when p is then used
- Possible to create more than one Parser
class ServiceV4 { // DO NOT USE  
    Parser parser = null;  
    Parser getParser() {  
        if (parser == null)  
            synchronized(this) {  
                if (parser == null)  
                    parser = new Parser();  
                }  
        parser = new Parser();  
        return parser;  
    }  
}  

Still possible to see parser as non-null, but not see parser’s fields initialized
Static Singletons

class ServiceV5 {
    static class ServiceParser {
        static Parser it = new Parser();
    }
    Parser getParser() {
        return ServiceParser.it;
    }
}

- Embed parser object in own class
  - Only useful for statics
- First use of a class forces class initialization
  - Later uses guaranteed to see class initialization
  - No explicit check needed
Isolation in Swing

- AWT thread owns all Swing components
  - No other thread may access them
- Eliminates need for locking
  - Still need care during initialization
- Can be fragile
  - Every programmer must obey rules
- But rules are usually easy to follow
  - Most Swing components accessed in handlers triggered within AWT thread
Accessing Isolated Objects

- Need safe inter-thread communication
- Swing uses runnable Event objects
  - Created by user thread
  - Serviced by AWT thread

```java
SwingUtilities.invokeLater(new Runnable(){
    public void run() {
        statusMessage.setText("Running");
    }
});
```
Some "Expert" issues

- Explicit memory barriers
  - For example: Only sometimes would you like a read of a certain reference to force fresh reads of referenced object fields
- CompareAndSwap (CAS)
  - Or transactional memory, etc
  - Most optimistic algorithms are not directly implementable otherwise.
Compliance

- It would be useful to check whether something claiming to be "thread safe" actually is.
- Safety means "bad things never happen"
  - Can only blindly check for a few bad things
    - Mainly surrounding data races
    - Even these are merely almost always bad
  - Others rely on programmer assertions and documentation
    - Leads to several hard problems...
Relative safety

- Writing code that is safe only within intended usage contexts is not a sin.
- But not documenting these intentions is.
- Documentation of such classes is hard work.

- **Unsafe** usually boils down to:
  
  A method or class imposes extra contextual side conditions that must hold for its intended post-conditions, effects, or invariants to hold.

- Even unsafe Java code preserves minimal safety properties: no wild pointers, etc

- Potential non-liveness defined similarly, but phrased in terms of progress guarantees
Sample policy constraints

- Ownership
  - Method $m$ is only called from Thread $t$.
  - Object $x$ is only accessed by Thread $t$.
  - Object $x$ is only accessed by Object $y$.

- Dynamic Exclusion
  - Either Object $x$ is only accessed by one thread, or current thread holds an exclusion lock covering $x$ for duration of method $m$.

- Ordering
  - Method $m_2$ is called only after method $m_1$
  - Lock $L_2$ is held only if lock $L_1$ is held

- Blocking
  - Callback method $m$ does not wait()
Specifying safety properties

- Need ways to explicitly state constraints, pre-conditions, etc
- Many fall under a few common categories
- A useful subset can be expressed via `assert`
- Potentially amenable to tool-based compliance checking
- But requires a more formal approach than most programmers are willing to undertake
- But concurrent components often have complicated specifications
Documenting constraints

➤ Failure–based approach:
  ➤ List specific race hazards, etc and their consequences
    ➤ List common errors that lead to them
    ➤ Not always possible due to polymorphism

➤ Constructive approach:
  ➤ Explain in documentation exactly how to meet constraints in common usage contexts.
  ➤ But not as helpful when people encounter uncommon contexts.
Conclusions

- A memory model can:
  - Clarify which low-synch idioms are effective
  - Enable more optimization
  - Make it easier to find mistakes
  - Enable better automated safety checks