


## Correct and Efficient Synchronization of Java™ Technology-based Threads

**Doug Lea and William Pugh**  
<http://gee.cs.oswego.edu>  
<http://www.cs.umd.edu/~pugh>

TS-754, Correct and Efficient Synchronization of Java Threads

## Audience


- Assume you are familiar with basics of Java™ technology-based threads (“Java threads”)
  - Creating, starting and joining threads
  - Synchronization
  - `wait` and `notifyAll`
- Will talk about things that surprised a lot of experts
  - Including us, James Gosling, Guy Steele, ... (others discovered many of these)



TS-754, Correct and Efficient Synchronization of Java Threads

## Java Thread Specification


- Chapter 17 of the Java Language Spec
  - Chapter 8 of the Virtual Machine Spec
- Very, very hard to understand
  - Not even the authors understood it
  - Has subtle implications
    - That forbid standard compiler optimizations
  - All existing JVMs violate the specification
    - Some parts should be violated



TS-754, Correct and Efficient Synchronization of Java Threads

## Safety Issues in Multithreaded Systems


- Many intuitive assumptions do not hold
- Some widely used idioms are not safe
  - Double-check idiom
  - Checking non-volatile flag for thread termination
- Can't use testing to check for errors
  - Some anomalies will occur only on some platforms
    - e.g., multiprocessors



TS-754, Correct and Efficient Synchronization of Java Threads

## Revising the Thread Spec


- Work is underway to consider revising the Java Thread Spec
  - <http://www.cs.umd.edu/~pugh/java/memoryModel>
- Goals
  - Clear and easy to understand
  - Foster reliable multithreaded code
  - Allow for high performance JVMs
- Will effect JVMs
  - And badly written existing code
    - Including parts of Sun's JDK



TS-754, Correct and Efficient Synchronization of Java Threads

## What to do Today?


- Guidelines we will provide should work under both existing and future thread specs
- Don't try to read the official specs
- Avoid corner cases of the thread spec
  - Not needed for efficient and reliable programs



TS-754, Correct and Efficient Synchronization of Java Threads


### Three Aspects of Synchronization

- Atomicity
  - Locking to obtain mutual exclusion
- 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

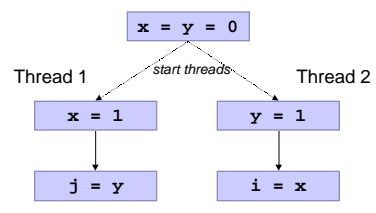


### Don't Be Too Clever


- People worry about the cost of synchronization
  - Try to devise schemes to communicate between threads
    - Without using synchronization
- Very difficult to do correctly
  - Inter-thread communication without synchronization is not intuitive



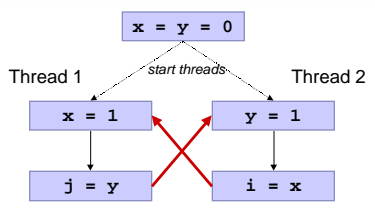
### Quiz Time




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 multi-processor, values not synchronized in global memory
- Must use synchronization to enforce visibility and ordering
  - As well as mutual exclusion

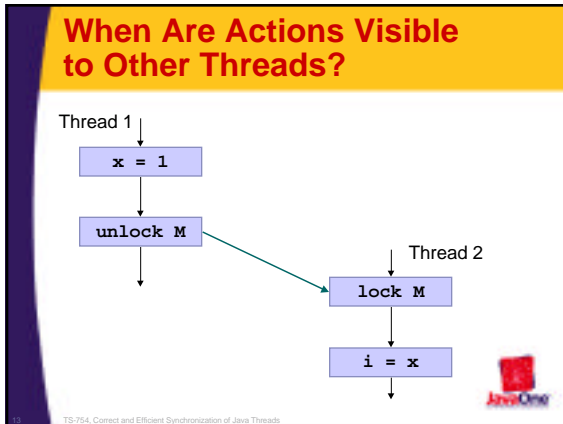


### Synchronization Actions (approximately)

```

// 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();
    
```





### What Does Volatile Mean?

- C/C++ spec
  - There is no implementation independent meaning of volatile
- Situation a little better with Java technology
  - Volatile reads/writes guaranteed to go directly to main memory
    - Can't be cached in registers or local memory

### Using Volatile

- Volatile used to guarantee visibility of writes
  - `stop` must be declared volatile
  - Otherwise, compiler could keep in register

```

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

```

### Using Volatile to Guard Other Fields Doesn't Work

- Do not use - Does not work

```

class Future {
    private volatile boolean ready = false;
    private Object data = null;
    public Object get() {
        if (!ready) return null;
        return data;
    }
    // only one thread may ever call put
    public void put(Object o) {
        data = o;
        ready = true;
    }
}

```

### Nobody Implements Volatile Correctly


- Existing JMM requires sequential consistency for volatile variables
  - In quiz example, if `x` and `y` are volatile
  - Should be impossible to see `i = 0` and `j = 0`
  - Haven't found any JVMs that enforce it
- Reads/writes of volatile longs/doubles
  - Guaranteed to be atomic (see old or new value, not a mixture)
- Some JVMs ignore volatile flag

### Volatile Compliance

	no compiler optimizations	sequential consistency	four memory barriers
2010-02-28 JDK 1.7.0_25 ENM	pass	fail	pass
2010-02-28 JDK 1.7.0_25 HotSpot Client	fail	fail	fail
2010-02-28 JDK 1.7.0_25 HotSpot Server	fail	fail	fail
2010-02-28 JDK 1.7.0_25 HotSpot Server	pass	fail	fail
2010-02-28 JDK 1.7.0_25 HotSpot Server	pass	fail	fail
2010-02-28 JDK 1.7.0_25 HotSpot Server	pass	fail	fail


### Why Use Volatile?

- Since the semantics are implemented inconsistently
- Future-proof your code
  - Prohibit optimizations compilers might do in the future
- Works well for flags
  - More complicated uses are tricky
- Revising the thread spec...
  - Test compliance
  - Strengthen to make easier to use




### Cost of Synchronization

- Few good public multithreaded benchmarks
  - See us if you want to help
- Volano Benchmark
  - Most widely used server benchmark
  - Multithreaded chat room server
  - Client performs 4.8M synchronizations
    - 8K useful (0.2%)
  - Server 43M synchronizations
    - 1.7M useful (4%)




### Synchronization in VolanoMark Client

7,684 synchronizations on shared monitors  
4,828,130 thread local synchronizations




### Cost of Synchronization in VolanoMark

- Removed synchronization of
  - java.io.BufferedInputStream
  - java.io.BufferedOutputStream
- Performance (2 processor Ultra 60)
  - Larger is better
  - HotSpot (1.3 beta)
    - Original: 4788
    - Altered: 4923 (+3%)
  - Exact VM (1.2.2)
    - Original: 6649
    - Altered: 6874 (+3%)




### Most Synchronization is on Thread Local Objects

- Synchronization on thread local object
  - Is useless
  - Current spec says it isn't quite a no-op
    - But very hard to use usefully
  - Revised spec will likely make it a no-op
- Largely arises from using synchronized classes
  - In places where not required




### Synchronize when Needed

- Places where threads interact
  - Need synchronization
  - Need careful thought
  - Need documentation
  - Cost of required synchronization not significant
    - For most applications
    - No need to get tricky
- Elsewhere, using a synchronized class can be expensive



### Synchronized Classes

- Some classes are synchronized
  - Vector, Hashtable, Stack
  - Most Input/Output Streams
- Contrast with 1.2 Collection classes
  - By default, not synchronized
  - Can request synchronized version
- Using synchronized classes
  - Often doesn't suffice for concurrent interaction




### Synchronized Collections Aren't Always Enough

- Transactions (DO NOT USE)

```


ID getID(String name) {
    ID x = (ID) h.get(name);
    if (x == null) {
        x = new ID();
        h.put(name, x);
    }
    return x;
}
    
```

- Iterators
  - Can't modify collection while another thread is iterating through it




### Concurrent Interactions

- Often need entire transactions to be atomic
  - Reading and updating a Map
  - Writing a record to an OutputStream
- OutputStreams are synchronized
  - Can have multiple threads trying to write to the same OutputStream
  - Output from each thread is nondeterministically interleaved
  - Essentially useless

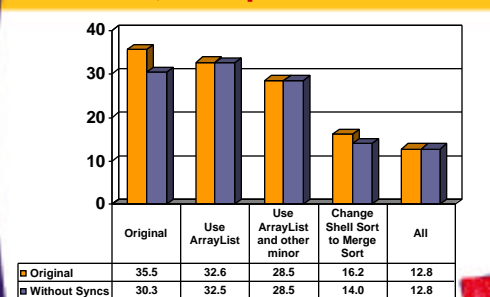


### Cost of Synchronization in SpecJVM DB Benchmark


- Program in the Spec JVM benchmark
- Does lots of synchronization
  - > 53,000,000 syncs
    - 99.9% comes from use of Vector
  - Benchmark is single threaded, all of it is useless
- Tried
  - Remove synchronizations
  - Switching to ArrayList
  - Improving the algorithm



### Execution Time of Spec JVM 209\_db, Hotspot Server




	Original	Use ArrayList	Use ArrayList and other minor	Change Shell Sort to Merge Sort	All
Original	35.5	32.6	28.5	16.2	12.8
Without Syncs	30.3	32.5	28.5	14.0	12.8




### Lessons

- Synchronization cost can be substantial
  - 10-20% for DB benchmark
  - Consider replacing all uses of Vector, Hashtable and Stack
- Use profiling
- Use better algorithms!
  - Cost of stupidity higher than cost of synchronization
  - Used built-in merge sort rather than hand-coded shell sort




### Designing Fast Code

- Make it right before you make it fast
- Avoid synchronization
  - Avoid sharing across threads
  - Don't lock already-protected objects
  - Use immutable fields and objects
  - Use volatile
- Avoid contention
  - Reduce lock scopes
  - Reduce lock durations



### Isolation in Swing

- Swing relies entirely on **isolation**
  - 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
  - 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 via runnable Event objects
    - Created by some other thread
    - Serviced by AWT thread

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




### GetX/SetX Access Methods

- Not synchronizing access methods
  - `int thermometer.getTemperature()`  
(doesn't work for references)
- Synchronizing access methods
  - `account.getTotalBalance()`
- Omitting access methods
  - `queue` doesn't need `getSize()`




### Things That Don't Work

- Double-Check Idiom
  - Also, unsynchronized reads/writes of refs
- Non-volatile flags
- Depending on sleep for visibility



### Initialization Check - v1 - OK


```
Basic version:
class Service {
    Parser parser = null;
    public synchronized void command() {
        if (parser == null)
            parser = new Parser(...);
        doCommand(parser.parse(...));
    }
    // ...
}
```



### Initialization checks - v2 - OK

Isolate check:


```
class ServiceV2 {
    Parser parser = null;
    synchronized Parser getParser() {
        if (parser == null)
            parser = new Parser();
        return parser;
    }
    public void command(...) {
        doCommand(getParser().parse(...));
    }
}
```



### Single-check - DO NOT USE

Try to do it without synchronization:


```
class ServiceV3 { // DO NOT USE
    Parser parser = null;
    Parser getParser() {
        if (parser == null)
            parser = new Parser();
        return parser;
    }
}
```



### Double-check - DO NOT USE


Try to minimize likelihood of synch:

```
class ServiceV4 { // DO NOT USE
    Parser parser = null;
    Parser getParser() {
        if (parser == null)
            synchronized(this) {
                if (parser == null)
                    parser = new Parser();
            }
        return parser;
    }
}
```




### Problems with Double-check

- Can reorder
  - Initialization of Parser object
  - Store into parser field
- ...Among other reasons
  - See JMM web page for gory details
- Can go wrong on uniprocessors
  - e.g., Symantic JIT
- Using volatile doesn't help
  - Under current JMM



### Alternatives to Double-Check

- Use synchronization
- Double check OK for primitive values
  - hashCode caching (still technically a data race)
- For static singletons
  - Put in separate class
  - First use of a class forces class initialization
  - Later uses guaranteed to see class initialization
  - No explicit check needed




### Rare Heavy New Objects

- Sometimes, need singleton that is expensive to create

```
static final Font HELVETICA
    = new FONT("Helvetica",Font.PLAIN, 24);

Font getFont() {
    if (!chinese)
        return HELVETICA;
    else
        return new ChineseFont();
}
```




### Using Static Singletons

```

static final Font HELVETICA
    = new Font("Helvetica",Font.PLAIN,24);

static class CFSingleton{
    static final Font CHINESE
        = new ChineseFont(...);
}
Font getFont() {
    if (!chinese)
        return HELVETICA;
    else
        return CFSingleton.CHINESE;
}
    
```




### Unsynchronized Reads/Writes of References

- Beware of unsynchronized getX/setX methods that return a reference
  - Same problems as double check
  - Doesn't help to synchronize *only* setX

```

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



### Thread Termination in Sun's Demo Applets

```


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.currentThread().sleep(delay);}
        catch (InterruptedException e) {}
        repaint();
    }
}
    
```


Annotations in the code:

- Red arrow pointing to `blinker = null;` in `stop()`: **unsynchronized access to blinker field**
- Red arrow pointing to `sleep(delay);` in `run()`: **confusing but not wrong: sleep is a static method**




### Problems

- Don't assume another thread will see your writes
  - Just because you did them
- Calling sleep doesn't guarantee you see changes made while you slept
  - Nothing to force thread that called stop to push change out of registers/cache



### Wrap-up

- Cost of synchronization operations can be significant
  - But cost of *needed* synchronization rarely is
- Thread interaction needs careful thought
  - But not too clever
- Need for synchronization...



### Wrapup - Synchronization

- Communication between threads
  - Requires both threads to synchronize
    - Or communication through volatile fields
- Synchronizing everything
  - Is rarely necessary
  - Can be expensive (3%-20% overhead)
  - May lead to deadlock
  - May not provide enough synchronization
    - e.g., transactions

