

The Secrets of Concurrency

Dr Heinz M. Kabutz

The Java Specialists' Newsletter

<http://www.javaspecialists.eu>



JavaSpecialists.EU

The Secrets of Concurrency

- **Make those multi-cores earn their keep ...**



Background

● Dr Heinz M. Kabutz

- ***German South African living in Europe***
- ***The Java Specialists' Newsletter***
 - 30 000 readers in 113 countries
- ***Actively coding Java***
- ***Teaching Java courses to companies:***
 - Java Patterns Course
 - Java 5 Delta Course
 - <http://www.javaspecialists.eu/courses>
- ***Java Champion***



Conference Fatigue

- **Time for confession**

- I struggle to concentrate listening to presentations
- Usually too difficult for me to understand
- Transition from basic to advanced happens too quickly

- **Lots of APIs, XML, Java demos that no one can read**

- `<demo usefulness="0">Hey, let's use a font size of 10! Don't you love this demo? I know you cannot read this, but showing off all this XML makes me feel so so clever <smiley /></demo>`

Structure of Talk

● **The Laws of Concurrency**

- **Law 1: The Law of the Ritalin Child**
- **Law 2: The Law of the Distracted Spearfisherman**
- **Law 3: The Law of the Overstocked Haberdashery**
- **Law 4: The Law of South African Crime**
- **Law 5: The Law of the Leaked Memo**
- **Law 6: The Law of the Corrupt Politician**
- **Law 7: The Law of the Micromanager**
- **Law 8: The Law of Greek Driving**
- **Law 9: The Law of Sudden Riches**
- **Law 10: The Law of the Uneaten Spinach**

The Law of the Ritalin Child

**Instead of suppressing
interruptions, deal with the cause.**

* Ritalin: A drug commonly prescribed to deal with naughty children.

Law 1: The Law of the Ritalin Child

- **Have you ever seen code like this?**

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException ex) {  
    // this won't happen here  
}
```

- **We will answer the following questions:**
 - What does `InterruptedException` mean?
 - How *should* we handle it?

Shutting Down Threads

● **Retrenchments**

- **Get rid of dead wood first!**

● **Shutdown threads when they are inactive**

- **Thread.sleep()**
- **BlockingQueue.get()**
- **Semaphore.acquire()**
- **wait()**
- **join()**

Thread “interrupted” Status

- **Thread “interrupted” == true**
 - **At next blocking code, InterruptedException is thrown**
 - **isInterrupted() returns true**
- **You can set “interrupted” status to true**

`someThread.interrupt();`
- **“interrupted” is set to false when**
 - **InterruptedException is thrown**
 - **Thread.interrupted() is called**
 - **Beware of side effect!**

How to Handle InterruptedException

- **Option 1: Simply re-throw** InterruptedException
 - Approach used by `java.util.concurrent`
 - Not always possible if we are overriding a method
- **Option 2: Catch it and return**
 - Our current “interrupted” state should be set to `true`

```
while (!Thread.currentThread().isInterrupted()) {  
    // do something  
    try {  
        TimeUnit.SECONDS.sleep(1000);  
    } catch (InterruptedException e) {  
        Thread.currentThread().interrupt();  
        break;  
    }  
}
```

The Law of the Distracted Spearfisherman

**Focus on one thread at a time. The
school of threads will blind you.**

* The best defense for a fish is to swim
next to a bigger, better fish.

Law 2: The Law of the Distracted Spearfisherman

- **You must understand what *every* thread is doing in your system**
 - **Good reason to have fewer threads!**
- **Don't jump from thread to thread, hoping to find problems**

Causing Thread Dumps

- **The jstack tool dumps threads of process**
 - **Similar to CTRL+Break (Windows) or CTRL+\ (Unix)**
- **For thread dump JSP page**
 - **<http://www.javaspecialists.eu/archive/Issue132.html>**
 - **Sorted threads allow you to diff between calls**

The Law of the Overstocked Haberdashery

Having too many threads is bad for your application. Performance will degrade and debugging will become difficult.

* Haberdasher: A shop selling sewing wares, e.g. threads and needles.

Law 3: The Law of the Overstocked Haberdashery

- **Story: Client-side library running on server**
- **We will answer the following questions:**
 - **How many threads can you create?**
 - **What is the limiting factor?**
 - **How can we create more threads?**

JRE Dies with Internal Error

```
Exception in thread "main" java.lang.OutOfMemoryError: unable
to create new native thread
  at java.lang.Thread.start0(Native Method)
  at java.lang.Thread.start(Thread.java:597)
  at ThreadCreationTest$1.<init>(ThreadCreationTest.java:8)
  at ThreadCreationTest.main(ThreadCreationTest.java:7)
#
# An unexpected error has been detected by Java Runtime
Environment:
#
# Internal Error (455843455054494F4E530E4350500134) #
# Java VM: Java HotSpot(TM) Client VM (1.6.0_01-b06 mixed
mode, sharing)
# An error report file with more information is saved as
hs_err_pid22142.log
#
Aborted (core dumped)
```


How to Create More Threads?

- **We created about 9000 threads**
- **Reduce stack size**
 - **java -Xss48k ThreadCreationTest**
 - **32284 threads**
 - **Had to kill with -9**
 - **My first computer had 48k total memory**
 - **Imagine 32000 ZX Spectrums connected as one computer!**
 - **Can cause other problems**
 - **See *The Law of the Distracted Spearfisherman***

How Many Threads is Healthy?

- **Additional threads should improve performance**
- **Not too many active threads**
 - 4 - 20 active per core
- **Inactive threads**
 - Number is architecture specific
 - But 9000 per core is way too much
 - Consume memory
 - Can cause sudden death of the JVM
- **Azul Systems (<http://www.azulsystems.com>)**
 - Build systems with 768 cores and 768GB memory
 - Can have thousands of *active* threads

Traffic Calming

- **Thread pooling good way to control number**
- **Use new ExecutorService**
 - Fixed Thread Pool
- **For lots of small tasks, thread pools can be faster**
 - Not main consideration

```
import java.util.concurrent.*;

public class ThreadConstructTest {
    private static final int NUMBER_OF_THREADS = 100 * 1000;

    private Semaphore semaphore = new Semaphore(10);
    private final Runnable job = new Runnable() {
        public void run() {
            semaphore.release();
        }
    };

    public void noThreadPool() throws InterruptedException {
        for (int i = 0; i < NUMBER_OF_THREADS; i++) {
            semaphore.acquire();
            new Thread(job).start();
        }
        // wait for all jobs to finish
        semaphore.acquire(10);
        semaphore.release(10);
    }
}
```

```
public void fixedThreadPool() throws InterruptedException {
    ExecutorService pool = Executors.newFixedThreadPool(12);
    for (int i = 0; i < NUMBER_OF_THREADS; i++) {
        semaphore.acquire();
        pool.submit(job);
    }
    semaphore.acquire(10); semaphore.release(10);
    pool.shutdown();
}
```

16004
842

```
public static void main(String[] args) throws Exception {
    ThreadConstructTest test = new ThreadConstructTest();

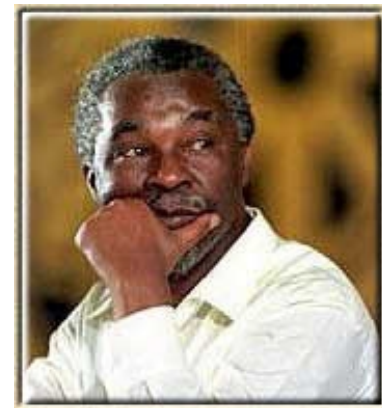
    long time = System.currentTimeMillis();
    test.noThreadPool();
    System.out.println(System.currentTimeMillis() - time);

    time = System.currentTimeMillis();
    test.fixedThreadPool();
    System.out.println(System.currentTimeMillis() - time);
}
}
```



The Law of South African Crime

You might miss important information if you try to be too clever.



* “Crime is a perception”

Law 4: The Law of South African Crime

- **Java Memory Model allows thread to keep local copy of fields**
- **Your thread might not see another thread's changes**
- **Usually happens when you try to avoid synchronization**

MyThread Might Never End!

```
public class MyThread extends Thread {  
    private boolean running = true;  
    public void run() {  
        while(running) {  
            // do something  
        }  
    }  
    public void shutdown() {  
        running = false;  
    }  
}
```


Making Field Changes Visible

- **Three ways of preventing this**
 - **Make field** volatile
 - **Make field** final **thus putting a “freeze” on value**
 - **Make read and writes to field** synchronized
 - **Also includes new locks**

Better MyThread

```
public class MyThread extends Thread {  
    private volatile boolean running = true;  
    public void run() {  
        while(running) {  
            // do something  
        }  
    }  
    public void shutdown() {  
        running = false;  
    }  
}
```

The Law of the Leaked Memo

**This may never happen, but when it
does, check your synchronization**

* Memo about hostile takeover bid left
lying in photocopy machine

Law 5: The Law of the Leaked Memo

- **If two threads call f() and g(), what are a and b equal to?**

```
public class EarlyWrites {  
    private int x;  
    private int y;  
  
    public void f() {  
        int a = x;  
        y = 3;  
    }  
  
    public void g() {  
        int b = y;  
        x = 4;  
    }  
}
```

**Early writes can result in:
a=4, b=3**

The order of Things

- **Java Memory Model allows reordering of statements**
- **Includes writing of fields**
- **To the writing thread, statements appear in order**

How to Prevent This?

- **JVM is not allowed to move writes out of synchronized **block****
 - **Allowed to move statements *into* a synchronized block**
- **Keyword `volatile` prevents early writes**
 - **From the Java Memory Model:**
 - **There is a happens-before edge from a write to a volatile variable `v` to all subsequent reads of `v` by any thread (where subsequent is defined according to the synchronization order)**

The Law of the Corrupt Politician

**In the absence of proper controls,
corruption is unavoidable.**

* Power corrupts. Absolute power
corrupts absolutely.

Law 6: The Law of the Corrupt Politician

- **Without controls, the best code can go bad**

```
public class BankAccount {  
    private int balance;  
    public BankAccount(int balance) {  
        this.balance = balance;  
    }  
    public void deposit(int amount) {  
        balance += amount;  
    }  
    public void withdraw(int amount) {  
        deposit(-amount);  
    }  
    public int getBalance() { return balance; }  
}
```


What happens?

- **The += operation is *not* atomic**
- **Thread 1**
 - Reads balance = 1000
 - Locally adds 100 = 1100
 - Before the balance written, Thread 1 gets swapped out
- **Thread 2**
 - Reads balance=1000
 - Locally subtracts 100 = 900
 - Writes 900 to the *balance* field
- **Thread 1**
 - Writes 1100 to the *balance* field

Solutions

- **Pre Java 5**

- **synchronized**
 - **Never use “this” as a monitor**

- **Java 5 and 6**

- **Lock, ReadWriteLock**
- **AtomicInteger – dealt with in *The Law of the Micromanager***

Pre-Java 5

```
public class BankAccount {  
    private int balance;  
    private final Object lock = new Object();  
    public BankAccount(int balance) {  
        this.balance = balance;  
    }  
    public void deposit(int amount) {  
        synchronized(lock) { balance += amount; }  
    }  
    public void withdraw(int amount) {  
        deposit(-amount);  
    }  
    public int getBalance() {  
        synchronized(lock) { return balance; }  
    }  
}
```

ReentrantLocks

- **Basic monitors cannot be interrupted and will never give up trying to get locked**
 - The Law of the Uneaten Spinach
- **Java 5 Locks can be interrupted or time out after some time**
- **Remember to unlock in a finally block**

```
private final Lock lock = new ReentrantLock();

public void deposit(int amount) {
    lock.lock();
    try {
        balance += amount;
    } finally {
        lock.unlock();
    }
}

public int getBalance() {
    lock.lock();
    try {
        return balance;
    } finally {
        lock.unlock();
    }
}
```

ReadWriteLocks

- **Can distinguish *read* and *write* locks**
- **Use `ReentrantReadWriteLock`**
- **Then lock either the write or the read action**
 - `lock.writeLock().lock();`
 - `lock.writeLock().unlock();`

```
private final ReadWriteLock lock =  
    new ReentrantReadWriteLock();  
  
public void deposit(int amount) {  
    lock.writeLock().lock();  
    try {  
        balance += amount;  
    } finally {  
        lock.writeLock().unlock();  
    }  
}  
  
public int getBalance() {  
    lock.readLock().lock();  
    try {  
        return balance;  
    } finally {  
        lock.readLock().unlock();  
    }  
}
```



The Law of the Micromanager

**Even in life, it wastes effort and
frustrates the other *threads*.**

* ***mi·cro·man·age***: to manage or control
with excessive attention to minor details.

Law 7: The Law of the Micromanager

- **Thread contention is difficult to spot**
- **Performance does not scale**
- **None of the usual suspects**
 - CPU
 - Disk
 - Network
 - Garbage collection
- **Points to thread contention**

Real Example – Don't Do This!

- **How to add contention**

```
String WRITE_LOCK_OBJECT = "WRITE_LOCK_OBJECT";
```

- **Later on in the class**

```
synchronized(WRITE_LOCK_OBJECT) { ... }
```

- **Constant Strings are flyweights!**
 - Multiple parts of the code locking on one object
 - Can also cause deadlocks and livelocks
- **Thanks to Java Specialists' Newsletter subscriber David Hallett for sharing this with me**

AtomicInteger

- **Thread safe without explicit locking**
- **Tries to update the value repeatedly until success**

```
public final int addAndGet(int delta) {  
    for (;;) {  
        int current = get();  
        int next = current + delta;  
        if (compareAndSet(current, next))  
            return next;  
    }  
}
```

```
import java.util.concurrent.atomic.AtomicInteger;

public class BankAccount {
    private final AtomicInteger balance =
        new AtomicInteger();

    public BankAccount(int balance) {
        this.balance.set(balance);
    }

    public void deposit(int amount) {
        balance.addAndGet(amount);
    }

    public void withdraw(int amount) {
        deposit(-amount);
    }

    public int getBalance() {
        return balance.intValue();
    }
}
```

Incorrect Code May Still Work

- **For a specific hardware architecture, ignore**
 - *The Law of South African Crime*
 - *The Law of the Leaked Memo*
- **Rather stick to well-written code**
 - **Atomics**
 - **ConcurrentHashMap**
 - **ConcurrentLinkedQueue**
 - **ConcurrentNavigableMap**
 - **ConcurrentSkipListMap and ConcurrentSkipListSet**

The Law of Greek Driving

**The JVM rules are not very strict.
Your code is probably wrong, even
if it works.**

* Don't ever *stop* at a stop sign if
you treasure your car!

Law 8: The Law of Greek Driving

- **Learn the JVM Rules !**
- **Example from JSR 133 – Java Memory Model**
 - **VM implementers are encouraged to avoid splitting their 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid this.**

JSR 133 allows this – NOT a Bug

- **Method set() called by two threads with**
 - 0x12345678ABCD0000L
 - 0x1111111111111111L

```
public class LongFields {  
    private long value;  
    public void set(long v) { value = v; }  
    public long get()      { return value; }  
}
```

- **Besides obvious answers, “value” could now also be**
 - 0x11111111ABCD0000L
 - 0x1234567811111111L

Java Virtual Machine Specification

- **Gives great freedom to JVM writers**
- **Makes it difficult to write 100% correct Java**
 - It might work on all JVMs to date, but that does not mean it is correct!
- **Theory vs Practice clash**

Synchronize at the Right Places

- **Too much synchronization causes contention**
 - As you increase CPUs, performance does not improve
 - The Law of the Micromanager
- **Lack of synchronization leads to corrupt data**
 - The Law of the Corrupt Politician
- **Fields might be written early**
 - The Law of the Leaked Memo
- **Changes to shared fields might not be visible**
 - The Law of South African Crime

Does This Code Work?

```
public class CASCounter implements Counter {
    private long count = 0;
    private Thread owner;
    public long getCount() { return count; }
    public void increment() {
        do {
            while (this.owner != null);    // wait
            this.owner = Thread.currentThread();
            for (int i = 0; i < 6; i++);    // delay
        } while (this.owner != Thread.currentThread());
        this.count++;
        this.owner = null;
    }
}
```

Answer: Yes, it Does!

- **At least, on the architectures we tried ...**
- **What could go wrong?**
 - **Variable owner might not get updated in your thread**
 - **Visibility problem**
 - **The Law of South African Crime**
 - **Can cause an infinite loop**

The Law of Sudden Riches

**Additional resources (faster CPU,
disk or network, more memory) for
seemingly stable system can make
it unstable.**

* Sudden inheritance or lottery win ...

Law 9: The Law of Sudden Riches

- **Better hardware can break system**
 - **Old system: Dual processor**
 - **New system: Dual core, dual processor**

Faster Hardware

- **Latent defects show up more quickly**
 - Instead of once a year, now once a week
- **Faster hardware often coincides with higher utilization by customers**
 - More contention
- **E.g. DOM tree becomes corrupted**
 - Detected problem by synchronizing all subsystem access
 - Fixed by copying the nodes whenever they were read

The Law of the Uneaten Spinach

A deadlock in Java can only be resolved by restarting the Java Virtual Machine.

* Imagine a stubborn father insisting that his stubborn daughter eat her spinach before going to bed

Law 10: The Law of the Uneaten Spinach

- **Part of program stops responding**
- **GUI does not repaint**
 - Under Swing
- **Users cannot log in anymore**
 - Could also be The Law of the Corrupt Politician
- **Two threads want what the other has**
 - And are not willing to part with what they already have

Using Multiple Locks

```
public class ProblemChild {
    private final Object lock = new Object();
    public synchronized void f() {
        synchronized(lock) {
            // do something ...
        }
    }
    public void g() {
        synchronized(lock) {
            f();
        }
    }
}
```

Finding the Deadlock

Pressing CTRL+Break or CTRL+\ or use jstack

Full thread dump:

Found one Java-level deadlock:

=====

"g()":

waiting to lock monitor 0x0023e274 (object
0x22ac5808, a com.maxoft.ProblemChild),
which is held by "f()"

"f()":

waiting to lock monitor 0x0023e294 (object
0x22ac5818, a java.lang.Object),
which is held by "g()"

Deadlock Means You Are ***Dead ! ! !***

- **Deadlock can be found with jconsole**
- **However, there is no way to resolve it**
- **Better to automatically raise critical error**
 - **Newsletter 130 – Deadlock Detection with new Lock**
 - **<http://www.javaspecialists.eu/archive/Issue130.html>**

Conclusion

- **Threading is easy, when you know the rules**
- **Tons of resources on JavaSpecialists.EU**
 - <http://www.javaspecialists.eu>

The Secrets of Concurrency

Dr Heinz M. Kabutz

The Java Specialists' Newsletter

<http://www.javaspecialists.eu>



JavaSpecialists.EU