Nebenläufigkeit und Java

Dr Heinz M. Kabutz

Javaspecialists.eu java training

Short Introduction to Speaker

- **Dr Heinz Kabutz**
  - Deutscher Südafrikaner, am Kap der guten Hoffenung geboren
    - Verbleibt jetzt auf der griechischen Insel Kreta
  - Schreibt den The Java Specialists' Newsletter
    - http://www.javaspecialists.eu/archive/archive.html
  - Gibt interessante Kurse über fortgeschrittenes Java Programmieren
  - Einer der ersten Java Champions
    - http://java-champions.java.net/

Fragen

- **Bitte Fragen fragen :-)**
  - **Zu jeder Zeit**
- **Dumme Fragen gibt es nicht**
  - **Nur die, die du nicht fragst**
- **Es macht es interessanter wenn ihr interaktiv mitmacht**

1.1: History of Concurrency

- **Concurrency allows better utilization of our hardware**
  - Whilst we are waiting for IO to complete, we can do something else
    - We call this "concurrent programming"
  - We can keep all our CPU cores busy
    - We call this "parallel programming"
- **The two topics are related, but not exactly the same**
  - In Java, we have a "concurrent mark sweep" garbage collector
    - Application threads run concurrently with GC threads
      - Shorter stop-the-world pauses, application more responsive
  - We also have "parallel" garbage collectors
    - These distribute the work over all the cores
    - GC is completed faster, but stop-the-world pause might be long

Javaspecialists.eu

Let's Go Fast Fast Fast

Let's Go Fast Fast Fast

- In 2000, Intel predicted 10GHz chips on desktop by 2011
  - http://www.zdnet.com/news/taking-chips-to-10ghz-and-beyond/96055

Let's Go Fast Fast Fast

- In 2000, Intel predicted 10GHz chips on desktop by 2011
  - http://www.zdnet.com/news/taking-chips-to-10ghz-and-beyond/96055
- Core i7 990x hit the market early 2011
  - 3.46GHz clock stretching up to 3.73 GHz in turbo mode
  - 6 processing cores
  - Running in parallel, we get 22GHz of processing power!

Let's Go Fast Fast Fast

- **In 2000, Intel predicted 10GHz chips on desktop by 2011**
  - **http://www.zdnet.com/news/taking-chips-to-10ghz-and-beyond/96055**
- **Core i7 990x hit the market early 2011**
  - **3.46GHz clock stretching up to 3.73 GHz in turbo mode**
  - **6 processing cores**
  - **Running in parallel, we get 22GHz of processing power!**
- **Japanese 'K' Computer June 2011**
  - **8.2 petaFLOPS**
    - **8 200 000 000 000 000 floating point operations per second**
    - **Intel 8087 was 30 000 FLOPS, 273 billion times slower**
  - **548,352 cores from 68,544 2GHz 8-Core SPARC64 VIIIfx processors**

Shared Java Heap Space

- **When last did you let your brother-in-law share your car?**
- **Threads share objects from Java heap space**
  - **Allows finer-grained communication than with processes**
  - **We need to control how memory is modified**
    - **Otherwise, our hard work can be overwritten**
  - **Underlying hardware can further increase chance of race conditions**
    - **L1/L2/L3 caches**
    - **Thread might get swapped to another core dynamically**

1.2: Vorteile der Nebenläufigkeit

1.2: Benefits of Threads

- **Threads make our lives as programmers easier:**
  - **Concurrent Programming**
    - **Independent sequential workflows are easier to write**
      - **Blocking vs non-blocking IO**
    - **During wait times our CPU can be kept busy**
  - **Parallel Programming**
    - **Multiple cores can solve many difficult tasks faster**
      - **Support with Java 7 Fork/Join**

Kurzes Beispiel

- **Wie schreibt man einen EchoServer am einfachsten?**
  - **Der EchoServer wiederholt alles was wir ihm schicken**
- **Ist es einfacher einen Thread pro User zu haben?**
- **Was ist vom Design her besser?**

Ein Thread pro Socket

```
public class BlockingEchoServer {
  public static void main(String[] args) throws Exception {
    ServerSocket server = new ServerSocket(9080);
    ExecutorService pool = Executors.newCachedThreadPool();
    while (true) {
      final Socket socket = server.accept();
      pool.submit(new Callable<Void>() {
        public Void call() throws Exception {
          InputStream in = socket.getInputStream();
          OutputStream out = socket.getOutputStream();
          int i;
          while ((i = in.read()) != -1) {
            out.write(i);
          }
          out.close(); in.close(); return null;
        }
      });
    }
  }
}
```

Ein Thread kommuniziert mit all den Sockets

```
public class NioServer implements Runnable {
  private final Selector selector;
  private final ByteBuffer readBuffer =
    ByteBuffer.allocate(8192);
  private final EchoWorker worker;
  private final Queue<ChangeRequest> changeRequests =
    new ConcurrentLinkedQueue<ChangeRequest>();

  private final ConcurrentMap<SocketChannel, Queue<ByteBuffer>>
    pendingData = new ConcurrentHashMap
      <SocketChannel, Queue<ByteBuffer>>();

  public NioServer(InetAddress hostAddress, int port,
                   EchoWorker worker)
      throws IOException {
    this.worker = worker;
    this.selector = initSelector(hostAddress, port);
  }
```

Noch mehr Kode ...

```
private Selector initSelector(InetAddress hostAddress,
                              int port)
    throws IOException {
  Selector socketSelector = Selector.open();

  ServerSocketChannel serverChannel =
    ServerSocketChannel.open();
  serverChannel.configureBlocking(false);

  InetSocketAddress isa =
    new InetSocketAddress(hostAddress, port);
  serverChannel.socket().bind(isa);

  serverChannel.register(socketSelector,
    SelectionKey.OP_ACCEPT);

  return socketSelector;
}
```

Und mehr ...

```
public void run() {
  while (true) {
    try {
      ChangeRequest change;
      while ((change = changeRequests.poll()) != null) {
        switch (change.type) {
          case ChangeRequest.CHANGEOPS:
            SelectionKey key =
              change.socket.keyFor(this.selector);
            key.interestOps(change.ops);
        }
      }

      this.selector.select();

      Iterator<SelectionKey> selectedKeys =
        this.selector.selectedKeys().iterator();
```

Und noch ...

```
        while (selectedKeys.hasNext()) {
          SelectionKey key = selectedKeys.next();
          selectedKeys.remove();
          if (!key.isValid()) {
            continue;
          }

          if (key.isAcceptable()) {
            this.accept(key);
          } else if (key.isReadable()) {
            this.read(key);
          } else if (key.isWritable()) {
            this.write(key);
          }
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
```

Wann hört es endlich auf?

```
private void accept(SelectionKey key) throws IOException {
  ServerSocketChannel serverSocketChannel =
    (ServerSocketChannel) key.channel();

  SocketChannel socketChannel = serverSocketChannel.accept();
  socketChannel.configureBlocking(false);

  Queue<ByteBuffer> newQueue =
    new ConcurrentLinkedQueue<ByteBuffer>();
  pendingData.put(socketChannel, newQueue);

  socketChannel.register(this.selector,
    SelectionKey.OP_READ);
}
```

Noch mehr ...

```
private void read(SelectionKey key) throws IOException {
  SocketChannel socketChannel = (SocketChannel) key.channel();
  this.readBuffer.clear();
  int numRead;
  try {
    numRead = socketChannel.read(this.readBuffer);
  } catch (IOException e) {
    numRead = -1;
  }
  if (numRead == -1) {
    key.cancel();
    socketChannel.close();
    pendingData.remove(socketChannel);
    return;
  }
  this.worker.processData(this, socketChannel,
    this.readBuffer.array(), numRead);
}
```

Und noch ...

```
public void send(SocketChannel socket, byte[] data) {
  Queue<ByteBuffer> queue = this.pendingData.get(socket);
  queue.add(ByteBuffer.wrap(data));

  this.changeRequests.add(
    new ChangeRequest(socket, ChangeRequest.CHANGEOPS,
      SelectionKey.OP_WRITE));

  this.selector.wakeup();
}
```

Fast sind wir da ...

```
private void write(SelectionKey key) throws IOException {
  SocketChannel socketChannel = (SocketChannel) key.channel();

  Queue<ByteBuffer> queue =
    this.pendingData.get(socketChannel);

  ByteBuffer buf;
  while ((buf = queue.peek()) != null) {
    socketChannel.write(buf);
    if (buf.remaining() > 0) {
      break;
    }
    queue.poll();
  }

  if (queue.isEmpty()) {
    key.interestOps(SelectionKey.OP_READ);
  }
}
```

Fertig!

```
  public static void main(String[] args) {
    try {
      EchoWorker worker = new EchoWorker();
      new Thread(worker).start();
      new Thread(new NioServer(null, 9090, worker)).start();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Doch nicht ganz?

```
class ServerDataEvent {
  public final NioServer server;
  public final SocketChannel socket;
  public final byte[] data;

  public ServerDataEvent(NioServer server,
                         SocketChannel socket,
                         byte[] data) {
    this.server = server;
    this.socket = socket;
    this.data = data;
  }
}
```

Wir brauchen noch einen EchoWorker ...

```
public class EchoWorker extends Thread {
  private final BlockingQueue<ServerDataEvent> queue =
    new LinkedBlockingQueue<ServerDataEvent>();

  public void processData(NioServer server,
                          SocketChannel socket,
                          byte[] data,
                          int count) {
    byte[] copy = new byte[count];
    System.arraycopy(data, 0, copy, 0, count);
    queue.add(new ServerDataEvent(server, socket, copy));
  }
```

Jetzt aber ...

```
  public void run() {
    while (true) {
      try {
        ServerDataEvent event = queue.take();
        event.server.send(event.socket, event.data);
      } catch (InterruptedException e) {
        break;
      }
    }
  }
}
```

Und einen ChangeRequest

```
public class ChangeRequest {
  public static final int REGISTER = 1;
  public static final int CHANGEOPS = 2;

  public final SocketChannel socket;
  public final int type;
  public final int ops;

  public ChangeRequest(SocketChannel socket,
                       int type, int ops) {
    this.socket = socket;
    this.type = type;
    this.ops = ops;
  }
}
```

Mit 1-1 Thread zu Socket war es eine Folie!

```
public class BlockingEchoServer {
  public static void main(String[] args) throws Exception {
    ServerSocket server = new ServerSocket(9080);
    ExecutorService pool = Executors.newCachedThreadPool();
    while (true) {
      final Socket socket = server.accept();
      pool.submit(new Callable<Void>() {
        public Void call() throws Exception {
          InputStream in = socket.getInputStream();
          OutputStream out = socket.getOutputStream();
          int i;
          while ((i = in.read()) != -1) {
            out.write(i);
          }
          out.close(); in.close(); return null;
        }
      });
    }
  }
}
```

Better Performance

- **Improved throughput**
  - **A program with only one thread can run on only one processor.**
    - **Single threaded in Adobe CS4, but uses multiple threads in CS5**
  - **On a 2-processor system, a single-threaded program is giving up access to 1/2 the available CPU's**
  - **On a 100-processor system, it is giving up access to 99%!**

Simpler Modeling

- It is easier to write code that does only one thing at a time
  - Simpler to code, easier to test, less bugs
- We can split up our tasks into separate threads
- Leads to cleaner abstractions

Responsive Programs

- **In Windows 3.1, formatting a floppy froze the machine**
  - **No preemptive multitasking possible**
- **If a task takes a while, our program must stay responsive**
  - **And it should always be possible to cancel the task**
- **Threads allow us to delegate jobs to thread pools**
  - **And to later fetch the results**

Gefahren der Threads

Risks of Threads

- **Threads have become too easy to create and use**

```
public class MyThread extends Thread {
  public void run() {
    // your concurrent task here
  }
}
MyThread thread = new MyThread();
thread.start();
```

- **Fortunately, frameworks try to stop us from creating them**
  - **But every single piece of Java code still is executed by a thread**
  - **We might see issues with contention, race conditions, deadlocks**

1.3.1. Safety Hazards

- **A program is thread safe if it functions correctly in a multi-threaded environment.**
- **Thread safety can be unexpectedly subtle.**
- **Ordering of operations in threads can be unpredictable.**
- **This is supposed to generate a unique int**

```
@NotThreadSafe
public class UnsafeSequence {
  private int value;
  public int getNext() {
    return value++; // Should return unique int
  }
}
```

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

public class UnsafeSequenceTest {
  public static void main(String[] args)
      throws InterruptedException {
    final Map uniqueNumbers = new ConcurrentHashMap();
    final UnsafeSequence seq = new UnsafeSequence();
    Runnable updater = new Runnable() {
      public void run() {
        for (int i = 0; i < 10000; i++) {
          int next = seq.getNext();
          String str = "value=" + next;
          uniqueNumbers.put(str, "dummy");
          System.out.println(str);
        }
      }
    };
    Thread t1 = new Thread(updater, "t1");
    Thread t2 = new Thread(updater, "t2");
    t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.println(uniqueNumbers.size());
  }
}
```

Warning: This program does not always show the race condition

ThreadSafe Sequence

- **By making getNext() synchronized, we get rid of the race condition**

```
@ThreadSafe
public class Sequence {
  @GuardedBy("this") private int value;
  public synchronized int getNext() {
    return value++;
  }
}
```

Threads gibt es überall

1.4: Threads are Everywhere

- **Even if you don’t create a thread, the framework might!**
- **Code called from these threads must be thread safe.**
- **Java threads are created by:**
  - The JVM
    - For housekeeping - garbage collection, finalization
    - And for running the main method
  - AWT & Swing
    - Event dispatch thread
  - Timer
  - Servlet Container, RMI, etc, etc, etc.

1.4: Threads are Everywhere

- **Nearly all Java apps are multithreaded**
  - **Concurrency is not an optional or advanced feature!**
- **You must understand thread safety!**

Timers

- **Concurrency services can cause application code to be called from threads not managed by the application:**

```
import java.util.*;

public class TimerTest {
  public static void main(String[] args) {
    TimerTask task = new TimerTask() {
      public void run() {
        System.out.println(new Date());
      }
    };
    Timer timer = new Timer();
    timer.schedule(task, 1000, 1000);
  }
}
```

Servlets and JavaServer Pages (JSPs)

- **HTTP request is sent to the appropriate servlet**
  - **JSPs are compiled to Servlets**
- **Servlets might be called simultaneously by many requests**
  - **Thus the servlet class itself must be thread safe**
- **Servlets often share data with other servlets**
  - **Via session or application scoped objects**
  - **Shared data must be thread safe**

Remote Method Invocation (RMI)

- **When RMI calls our remote object, it uses its own thread**
  - How many threads does RMI create?
  - Could the same remote method on the same remote object be called simultaneously in multiple RMI threads?
- **A remote object must guard against two safety hazards:**
  - Access to state that may be shared with other objects
  - Access to the state of the remote object itself
- **Like servlets, the same RMI object can be called by several threads at once**

Swing and AWT

- **You should only ever do things to the GUI from the Swing thread by passing your "job" to the AWT toolkit:**

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

- **Don't need this in GUI event handler methods (e.g. actionPerformed); they're already called by the Swing thread**

Thread Safety

2: Thread Safety

- **Man sollte sich an die Regeln halten**

Shared Data in Multi-Threading

- Multi-*processing* typically does not allow sharing of data
  - Inter-process communication is expensive
  - Limits the opportunities for parallelism
- Multi-*threading* allows sharing of fields or data
  - Communication between threads can be more light-weight
  - Can cause race conditions and data races
  - Thread safety is about managing access to shared, mutable state
- Each thread has his own stack memory
  - Contains call stack, local variables and parameters
  - We never have to worry about thread safety with these

Synchronization

- **Shared data needs to have correct *synchronization***
  - **This is not only done with** synchronized **keyword**
  - **Also includes volatile, atomics and explicit locks**

Your Program Probably Has Latent Defects

- **Your program *is* broken if several threads access shared data without correct synchronization**
  - **Even if it appears to work**
  - **You might have just been lucky so far**
- **We can fix it in three ways:**
  - **Stop sharing state across threads**
  - **Make state immutable**
  - **Use synchronization whenever**
- **Rather think of this up-front**
  - **Retrofitting can be hard work and error prone**

Object Orientation and Synchronization

- **Well encapsulated classes are easier to synchronize**
  - **Try to always make all data private**
  - **Try to make classes immutable**
  - **Document your assumptions**
    - **Check your pre-conditions and post-conditions**
    - ***Data races* can fool you, for example**

```
public void birthday() {
    int currentAge = age;
    age = age + 1;
    assert age == currentAge + 1;
}
```

Object Orientation and Synchronization

- **Well encapsulated classes are easier to synchronize**
  - **Try to always make all data private**
  - **Try to make classes immutable**
  - **Document your assumptions**
    - **Check your pre-conditions and post-conditions**
    - ***Data races* can fool you, for example**

```
public void birthday() {
    int currentAge = age;
    age = age + 1;
    assert age == currentAge + 1;
}
```

In Server HotSpot, assertion might never ever fail!

Locking

Thread Safety

Locking

- **Java allows us to define mutually exclusion locks**
  - **Called *mutexes***
- **With locking, we can make our classes thread-safe**
  - **However, we can also introduce contention**
    - **Performance killer that reduces parallelism**

Why is this @NotThreadSafe?

```
@NotThreadSafe
public class UnsafeCachingFactorizer
    implements Servlet {
  private final AtomicReference<BigInteger>lastNumber =
    new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors =
    new AtomicReference<BigInteger[]>();
  public void service(ServletRequest req,
                      ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, lastFactors.get());
    else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, factors);
    }
  }
}
```

UnsafeCachingFactorizer Errors

- **AtomicReferences are individually thread-safe**
  - **However, the two values** lastNumber **and** lastFactors **could point to different factors**
- **We need to update related state variables in a single atomic operation**
  - **Otherwise we have no guarantee that state is consistent**

Intrinsic Locks

- **Java has a built-in locking mechanism for atomicity**
  - **Also enforces visibility - next section**

```
synchronized (lock) {
  // Access or modify shared state guarded by lock
}
```

- **We can use *any* Java object as a lock**
  - **These are called *monitor* or *intrinsic* locks**
- **Locking and unlocking is automatically done by JVM**
  - **We can never "by mistake" forget to unlock an intrinsic lock**

What is Wrong with this Factorizer?

```
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber;
  @GuardedBy("this") private BigInteger[] lastFactors;
  public synchronized void service(ServletRequest req,
                                   ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber))
      encodeIntoResponse(resp, lastFactors);
    else {
      BigInteger[] factors = factor(i);
      lastNumber = i;
      lastFactors = factors;
      encodeIntoResponse(resp, factors);
    }
  }
}
```

What is Wrong with this Factorizer?

```
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber;
  @GuardedBy("this") private BigInteger[] lastFactors;
  public synchronized void service(ServletRequest req,
                                   ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber))
      encodeIntoResponse(resp, lastFactors);
    else {
      BigInteger[] factors = factor(i);
      lastNumber = i;
      lastFactors = factors;
      encodeIntoResponse(resp, factors);
    }
  }
}
```

A non-random load test might prove that this is very fast.

What is Wrong with this Factorizer?

```
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber;
  @GuardedBy("this") private BigInteger[] lastFactors;
  public synchronized void service(ServletRequest req,
                                   ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber))
      encodeIntoResponse(resp, lastFactors);
    else {
      BigInteger[] factors = factor(i);
      lastNumber = i;
      lastFactors = factors;
      encodeIntoResponse(resp, factors);
    }
  }
}
```

But with real data, contention would be a performance killer.

A non-random load test might prove that this is very fast.

Conclusion

Nebenläufigkeit

Nebenläufigkeit

- **Java unterstützt Threads und sie werden auch viel angewandt**
- **Wir müssen unsere Objekte absichern**
- **Dabei müssen wir "Contention" vermeiden**

Weitere Informationen

- **Java Specialists Master Kurs**
  - **Düsseldorf, 22-25 August 2011**
    - **Erster "Sommerkurs" in Deutschland**
    - **Mal sehen ob euch der lange Sommer langweilt :-)**
    - **http://www.javaspecialists.eu/courses/master.jsp**

Nebenläufigkeit und Java

Sonstige fragen?