Java im Multicore Zeitalter

(Programmiersprachen im Multicore-Zeitalter)

von Ramon Ising

Übersicht

1. Was ist Java?
2. Threads in Java
3. Java Memory Modell
4. Beispiele
5. Fork/Join Framework
6. ExecutorService
7. Fazit

1. Was ist Java?

- Objektorientierte Programmiersprache
- Aktuelle Version 7
- Java Development Kit (JDK)
- Java Runtime Enviroment (JRE)
  - Java Virtual Machine (JVM)
  - Platformunabhänigkeit

2. Threads in Java
Threads erstellen:

- Interface „Runnable“

```
class CounterCommand implements Runnable
{
  @Override public void run()
  {
    for ( int i = 0; i < 20; i++ )
      System.out.println( i );
  }
}
```

```
....
Thread t2 = new Thread( new CounterCommand() );
t2.start();

....
```

2. Threads in Java
Threads erstellen:

- Klasse „Thread“

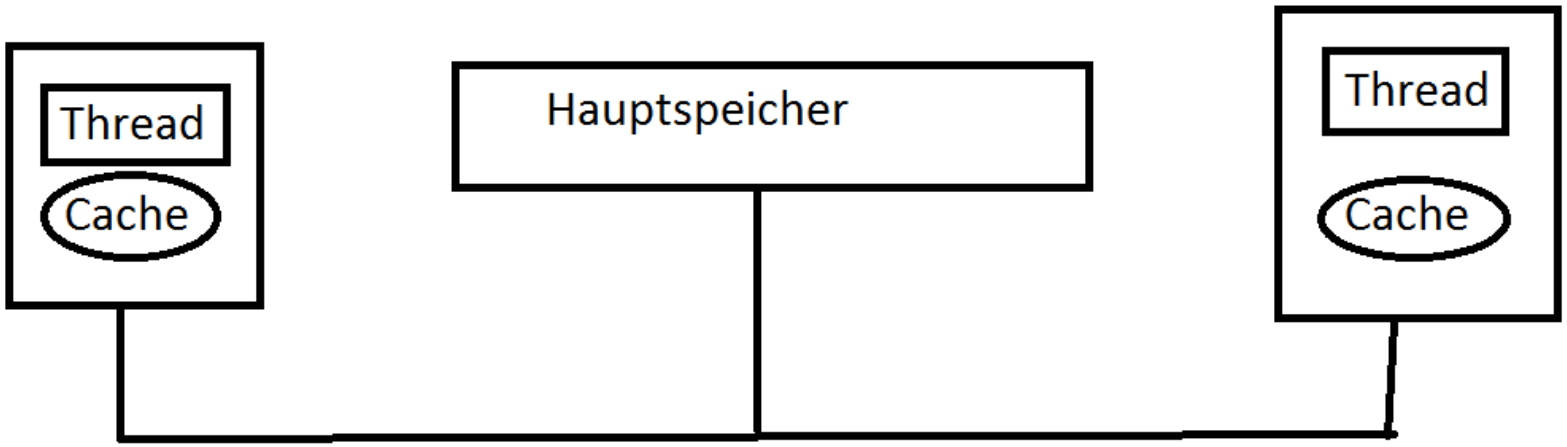
```
public class DateThread extends Thread
{
  @Override public void run()
  {
    for ( int i = 0; i < 20; i++ )
      System.out.println( new Date() );
  }
}

Thread t = new DateThread();
t.start();
```

2. Threads in Java
Eigenschaften

- getName() – setName()
- Thread.sleep() (InterruptedException)
- join()
- setDaemon()
- Interrupt() - isInterrupted()

3. Java Memory Modell



4. Beispiel

```
3  class Counter {
4      private int c = 0;
5
6      public void increment() {
7          c++;
8      }
9
10     public void decrement() {
11         c--;
12     }
13
14     public int value() {
15         return c;
16     }
17
18 }
```

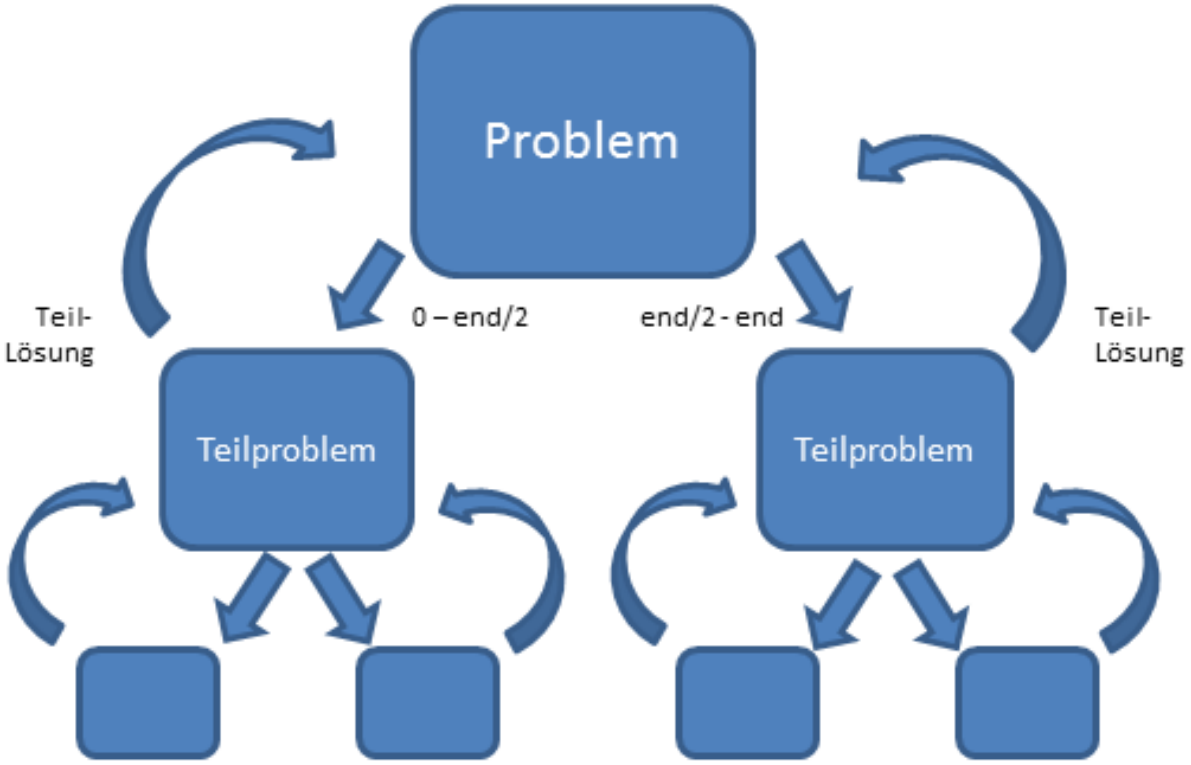
4. Beispiel

```
2
3  class SynchronizedCounter {
4      private int c = 0;
5
6      public synchronized void increment() {
7          c++;
8      }
9
10     public synchronized void decrement() {
11         c--;
12     }
13
14     public synchronized int value() {
15         return c;
16     }
17
18 }
19
```

4. Beispiel

```
4  import java.util.concurrent.atomic.AtomicInteger;
5
6  class AtomicCounter {
7      private AtomicInteger c = new AtomicInteger(0);
8
9      public void increment() {
10         c.incrementAndGet();
11     }
12
13     public void decrement() {
14         c.decrementAndGet();
15     }
16
17     public int value() {
18         return c.get();
19     }
20
21 }
22
```

5. Fork/Join Framework



5. Fork/Join Framework

```
public class Solver extends RecursiveAction {
    ...
    public Solver(int[] array, int lo, int hi) {
        this.list = array;
        this.lo = lo;
        this.hi = hi;

    }

    @Override
    protected void compute() {

        ....
       int anzahl1 = j - lo;
       int anzahl2 = hi - i;
       if(anzahl1>MINSIZE || anzahl2>MINSIZE){
           Solver s1 = new Solver(list,lo,j);
           Solver s2 = new Solver(list,i,hi);
           invokeAll(s1,s2);
       }else{
           Solver s1 = new Solver(list,lo,j);
           Solver s2 = new Solver(list,i,hi);
           s1.compute();
           s2.compute();
       }


   }

   public int[] getList() {
       return list;
   }
```

```
int[] liste = {4,8,1,2};
int nThreads = Runtime.getRuntime().availableProcessors();
Solver solv = new Solver(liste,0,liste.length-1);
ForkJoinPool pool = new ForkJoinPool(nThreads);
pool.invoke(solv);
```

6. ExecutorService

```
public class MyCallable implements Callable<Long> {
        @Override
        public Long call() throws Exception {
                long sum = 0;
                for (long i = 0; i <= 100; i++) {
                        sum += i;
                }
                return sum;
        }

}
```

6. ExecutorService

```
public class CallableFutures {
        private static final int NTHREDS = 10;

        public static void main(String[] args) {

                ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
                List<Future<Long>> list = new ArrayList<Future<Long>>();
                for (int i = 0; i < 20000; i++) {
                        Callable<Long> worker = new MyCallable();
                        Future<Long> submit = executor.submit(worker);
                        list.add(submit);
                }
                long sum = 0;
                System.out.println(list.size());
                // Now retrieve the result
                for (Future<Long> future : list) {
                        try {
                                sum += future.get();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        } catch (ExecutionException e) {
                                e.printStackTrace();
                        }
                }
                System.out.println(sum);
                executor.shutdown();
        }
}
```

7. Fazit

- Performances
  - Parallelverarbeitung in Java nicht optimal
  - Atome Variablen statt Synchronized-Block
- ExecutorService und Fork/Join Modell
  - Sehr einfache Programmierung
  - Threadverwaltung übernimmt Framework
- Bei Public-Variablen evtl. auf „volitale“ achten

Vielen Dank für Ihre Aufmerksamkeit!

Fragen?