Threads

Hans Jürgen Ohlbach

23. November 2017

Keywords: Threads

Empfohlene Vorkenntnisse: Prozesse, Java, Client-Server Kommunikation

Inhaltsverzeichnis

1 Einleitung und Motivation **2**

1.1 Eine Analogie: Küchen 2

2 Threads **3**

2.1 User-Level und Kernel-Level Threads 4

2.2 Vorteile des Threadkonzepts 5

3 Parallelisierte Algorithmen mit Threads **5**

3.1 Beispiel: Fakultätsberechnung 6

3.2 Beispiel: Paralleles Bubblesort 9

3.3 Serverprogrammierung 10

4 Race Conditions in Java **12**

4.1 Monitore 12

4.2 volatile 14

5 Zusammenfassung **15**

1 Einleitung und Motivation

Moderne Betriebssysteme sind darauf ausgelegt, viele Prozesse quasi gleichzeitig[1] arbeiten zu lassen. Das Konzept passt sehr gut, wenn in allen Prozessen unterschiedliche Programme laufen. Es gibt jedoch Situationen, wo eine brauchbare Benutzerinteraktion nur dann zustande kommt, wenn sich ein Programm in viele Prozesse aufspaltet. Das klassische Beispiel ist ein Webserver, der eine unvorhersehbare Anzahl von Clients bedienen soll. Das ist im Prinzip machbar mit einem einzigen Prozess. Die Konsequenz ist allerdings, dass die Clients alle nur nacheinander bedient werden können. Nur ein Client ist zu jederzeit mit dem Server verbunden. Alle anderen müssen warten, bis dieser fertig ist, auch wenn er für längere Zeit überhaupt keine Interaktion macht, weil z.B. der Benutzer sich gerade einen Kaffee kocht.

Eine erste Lösung ist, für jeden Client eine eigene Kopie des Prozesses zu „forken“. Die Kopien können dann unabhängig voneinander parallel die Clients bedienen. Durch *virtuelle Speicherverwaltung* und *Copy on Write* wird beim Forken nicht wirklich der ganze Prozess kopiert, sondern nur die Prozesstabelle, die Seitentabelle, und die Teile im gemeinsam genutzten Arbeitsspeicher, die von einem der Prozesse verändert werden.

Das funktioniert, hat aber eine Reihe von Nachteilen:

- Auch bei virtueller Speicherverwaltung und Copy on Write verursacht das Forken eines Prozesses einen nicht unerheblichen Aufwand.
- Um den Eindruck echter Parallelität zu wecken, müssen die geforkten Prozesse häufig gewechselt werden. Prozesswechsel verursacht ebenfalls einen nicht unerheblichen Aufwand.
- Die Kommunikation zwischen geforkten Prozessen geht nur über Pipes, Sockets, Files, Interrupts oder vielleicht noch über Systemvariablen. All dies ist sehr umständlich und kostet Zeit.

Beim Beispiel des Webservers ist es ja so, dass alle geforkten Prozesse das gleiche Programm abarbeiten, nur jeweils an anderen Stellen im Programm, und mit anderen Daten. Daher hat man sich gefragt, ob es nicht machbar ist, innerhalb eines Prozesses gleichzeitig an verschiedenen Stellen des Programms zu arbeiten. Man bräuchte dazu eigentlich nur mehrere unabhängige Programmzähler. Wenn man zwischen den Programmzählen hin- und herschaltet, sieht es nach außen so aus, als ob mehrere unterschiedliche Programme gleichzeitig laufen. Noch besser wäre es, wenn jeder Programmzähler einem eigenen Prozessorkern zugeteilt wird. Dann hätte man echte Parallelität. Aus diesen Ideen ist das Konzept der *Threads* entstanden.

1.1 Eine Analogie: Küchen

Man könnte Prozesse mit Küchen vergleichen. Mehrere parallel laufende Prozesse entsprächen mehreren Küchen, mit jeweils einem eigenen Koch. Sie kochen unterschiedliche Speisen nach unterschiedlichen Rezepten. Manchmal kommunizieren sie miteinander, wenn sie sich z.B. in der Speisekammer treffen.

[1] Dadurch dass sich die Prozesse im Millisekundenbereich abwechseln, kommt auch bei Computern mit nur einem Prozessor der Eindruck von echter Parallelität zustande.

Die Analogie zu Threads wäre eine einzige Küche (der Prozess), aber mit mehreren Köchen, die aber nach *einem einzigen* Rezept (dem Programm) kochen. Sie können z.B. mehrere Portionen nach dem gleichen Rezept kochen. Sie können auch unterschiedliche Teile des Rezepts parallel kochen, einer brät z.B. das Fleisch, und ein anderen macht die Soße. Sie müssen sich aber die Geräte in der Küche teilen, und daher absprechen, wer Zugriff auf welches Gerät hat. Das wäre bei verschiedenen Küchen (Prozessen) nicht notwendig, es sei denn sie brauchen Dinge von außerhalb der Küche, z.B. eine neue Gasflasche für den Gasherd.

2 Threads

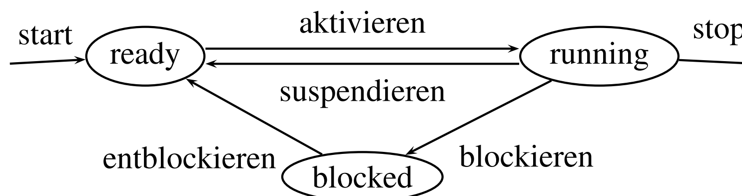
Mit einem *Prozess* assoziiert man i.A.:

- den Adressraum, den der Prozess benutzen kann,
- einen Programmzähler,
- den Zustand, in dem sich der Prozess gerade befindet,
- ein Stack, auf dem Daten beim Unterprogrammaufruf abgelegt werden,
- offene Dateien, E/A Geräten,
- Zugriffsrechte.

Im Multithreading-Betrieb gibt es weiterhin einen gemeinsamen Adressraum für den Prozess, mit all seinen Threads. Auch die Zugriffsrechte auf externe Ressourcen, z.B. Dateien sind für den Prozess als Ganzes definiert, da der Prozess einen Eigentümer hat, nicht aber die Threads.

Einige Ressourcen vervielfachen sich aber:

- Jeder Thread hat einen eigenen Programmzähler.
- Threads können und dürfen nicht permanent rechnen. Sie können blockiert sein, wenn sie auf externe Daten warten. Falls nicht genügend Prozessorkerne zur Verfügung stehen, werden sie auch regelmäßig suspendiert, damit andere Threads zum Zuge kommen können. Daher brauchen sie auch einen eigenen *Zustand*:



- Da Threads natürlich auch Unterprogramme aufrufen, brauchen sie auch ein eigenes Stack.
- Jeder Thread kann auch selbst Speicher anfordern. Daher braucht er auch einen Teil des Adressraumes für sich.

2.1 User-Level und Kernel-Level Threads

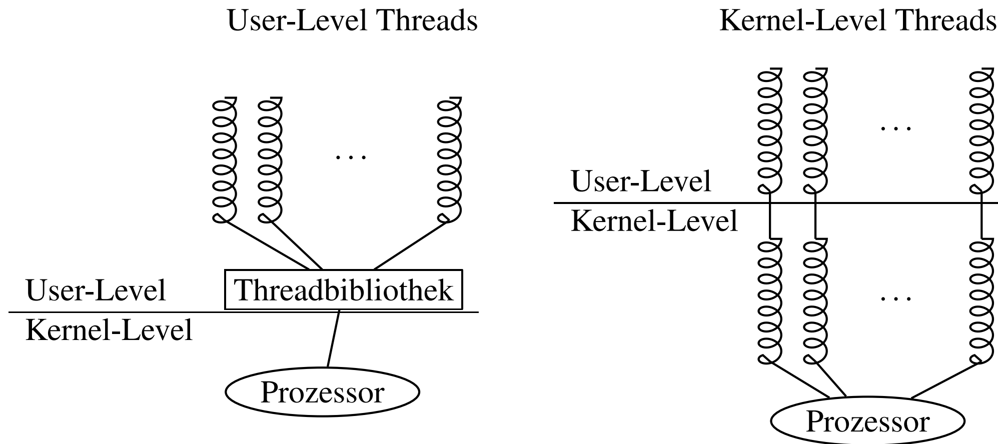
Die ersten Thread-Implementierungen wurden noch nicht vom Betriebssystem unterstützt. Daher musste die ganze Thread-Verwaltung im Prozess selbst gemacht werden. Der Prozess musste dann selbständig zwischen den Threads hin- und herschalten. Allerdings konnte er nicht mitbekommen, wenn ein gerade aktiver Prozess blockiert war, weil er z.B. auf externe Daten wartete. Das führte dazu, dass das Betriebssystem den ganzen Prozess blockiert hat, auch wenn andere Threads hätten weiter rechnen können.

Darüber hinaus kam keine echte Parallelität zustande, auch wenn mehrere Rechenkerne zur Verfügung standen. Da das Betriebssystem nur den Prozess als Ganzes sehen konnte, konnte es auch nur den Prozess als Ganzes einem Rechenkern zuordnen.

Mit der Weiterentwicklung der Betriebssysteme kam auch eine Unterstützung für die Threads hinzu. Seither verwaltet das Betriebssystem die Threads, mit zwei großen Vorteilen:

- wenn ein Thread blockiert ist, kann das Betriebssystem einen anderen Thread aktivieren,
- das Betriebssystem kann die Threads unterschiedlichen Prozessorkernen zuordnen.
  Damit erreicht man echte Parallelität.

Seither unterscheidet man *User-Level Threads* und *Kernel-Level Threads*:



2.2 Vorteile des Threadkonzepts

Das Konzept der Threads hat viele Vorteile:

- Zur Generierung eines neuen Threads in einem existierenden Prozess ist wesentlich weniger Zeit notwendig, als zur Generierung eines neuen Prozesses, teilweise ein Faktor 10!

  Beispiel: File-Server eines LANs
  - Für jede Dateianfrage genügt es, anstelle eines neuen Prozesses, einen neuen Thread zu generieren.
  - In kurzen Zeitabständen werden viele Threads generiert und terminiert.
  Noch effizienter ist es, wenn man nicht jedesmal einen neuen Thread erzeugt, sondern die Threads, die mit ihrer Arbeit fertig sind, in einem *Thread Pool* tut. Wird ein neuer Thread gebraucht, muss man nur dann einen neuen erzeugen, wenn der Thread Pool leer ist. Ein Thread aus dem Thread Pool kann dann auch die neue Aufgabe übernehmen.

- Im Gegensatz zu Prozessen können Threads untereinander kommunizieren, z.B. über globale Variablen, ohne den Betriebssystem-Kern zu involvieren. Das ist deutlich effizienter als eine Kommunikation über Pipes oder Sockets.

- Kommt es zu Blockierungen, so entsteht weniger Wartezeit, wenn nur einzelne Threads blockiert sind und andere abgearbeitet werden können.

- Threadwechsel innerhalb eines Prozesses erfordern weniger Zeit als Prozesswechsel.

- Algorithmen können echt parallelisiert werden, um die Möglichkeiten der Mehrkernrechner auszunutzen. Das ist mit Threads viel leichter zu implementieren, als mit Prozessen.

3 Parallelisierte Algorithmen mit Threads

Seit den ersten Computern, die mehr als einen Prozessor haben, hat man versucht, Algorithmen so umzustrukturieren, dass sie parallel arbeiten können. Mit der Einführung von Mikroprozessoren mit

mehr als einem Rechenkern hat nahezu jeder Computer die Möglichkeit, parallel zu rechnen. Damit ist es noch wichtiger geworden, Algorithmen zu parallelisieren, um diese Möglichkeiten auch zu nutzen. Die Programmierung solcher paralleler Algorithmen ist mit der Einführung des Thread-Konzepts deutlich einfacher geworden, so dass inzwischen jeder, der einen Computer hat, mit parallelisierten Algorithmen experimentieren kann. Da serielle Algorithmen inzwischen nur einen Bruchteil der Rechenkapazität moderner Computer ausnutzen, ist die Parallelisierung sogar unumgänglich, wenn es um die Effizienz von Programmen geht.

Der erste Schritt, um einen Algorithmus zu parallelisieren, ist natürlich, einen Weg zu finden, die Rechenschritte in möglichst unabhängige Teile aufzuspalten. Möglichst unabhängig bedeutet dabei, dass die Kommunikation zwischen den Teilen minimal sein sollte. Kommunikation bedeutet meistens, dass sich die Teile synchronisieren müssen, was oft Wartezeiten bedingt. Die einzelnen Teile sollten auch möglichst ausbalanciert sein, d.h. gleich viel Rechenlast bewältigen müssen.

3.1 Beispiel: Fakultätsberechnung

Wir illustrieren die Parallelisierung eines Algorithmus zunächst an der Berechnung der Fakultätsfunktion: $n! = 1 \cdot 2 \cdot \ldots \cdot n$. An diesem Beispiel kann man auch die Verwendung von Threads in Java einführen.

Angenommen, wir wollen 1000! mit 2 Threads berechnen. Eine erste Idee wäre: der erste Thread berechnet $1 \cdot 2 \cdot \ldots \cdot 500$, und der zweite Thread berechnet $501 \cdot 502 \cdot \ldots \cdot 1000$. Am Ende werden die beiden Ergebnisse noch multipliziert.

Warum ist das keine gute Idee?

Die Zahlen werden so groß, dass normale Integerzahlen nicht ausreichen, sondern man braucht einen Zahlentyp, der mit beliebig langen Integern umgehen kann. Bei solchen Zahlentypen ist aber die Komplexität der arithmetischen Operationen von der Größe der Zahl abhängig. Je größer die Zahl ist, desto länger brauchen Addition und Multiplikation. Die Multiplikation von $501 \cdot 502 \cdot \ldots \cdot 1000$ erzeugt aber deutlich größere Zahlen, und dauert daher viel länger, als die Multiplikation von $1 \cdot 2 \cdot \ldots \cdot 500$. Daher sind die beiden Teile nicht ausbalanciert.

Eine bessere Idee, die zu ausbalancierten Teilen führt, ist: der erste Thread multipliziert die geraden Zahlen, und der zweite Thread die ungeraden. Hat man n Threads, dann kann jeder davon Zahlen mit der Schrittweite n multiplizieren. Für $n = 8$, z.B. multipliziert der erste $1 \cdot 9 \cdot 17 \ldots$, der zweite multipliziert $2 \cdot 10 \cdot 18 \ldots$, usw. Am Ende werden die n Ergebnisse noch miteinander multipliziert.

Fakultätsberechnung mit Java: Das folgende Programm berechnet 500000!, zunächst nur seriell in einer einfachen Schleife. Es benutzt dazu die Java Klasse BigInteger, welche Integerzahlen beliebiger Länge zur Verfügung stellt.

```
import java.math.BigInteger;

public class FakultaetSeriell {
    public static void main(String[] args) {

        int n = 500000;
        BigInteger fakultaet = BigInteger.ONE;

        for(int i = 2; i <= n; ++i) {
            fakultaet = fakultaet.multiply(BigInteger.valueOf(i));}

        System.out.println(fakultaet);
    }
}
```

Um in Java Threads zu programmieren, gibt es mehrere Möglichkeiten. Eine davon geht folgendermaßen:

- Definiere eine Unterklasse der Java-Klasse Thread, und dort eine Methode `public void run(){...}`.
  Die run-Methode macht die Berechnung. Die Klasse kann beliebige Instanz- und Klassenvariablen, sowie beliebige weitere Methoden haben.
- Jetzt kann man mehrere Instanzen dieser Klasse mit `new` erzeugen. Dies startet aber noch keine Threads.
- Die Threads werden mit der Methode `start()` gestartet.

Das folgende Programm geht nach diesem Schema vor und berechnet wieder 500000!. Die Klasse `Fakultaet` als Unterklasse von Thread hat eigene Variablen `start` und `schritte`, so dass jeder Thread eigene Versionen dieser Variablen bekommt. In einer eigenen Variable `fakultaet` speichert der Thread dann seine Zwischenergebnisse.

In der `main`-Methode wird zunächst mit `Runtime.getRuntime().availableProcessors();` die Anzahl der Prozessoren im Rechner bestimmt. Danach richtet sich die Anzahl der Threads.

Dann werden die Threads erzeugt, im Array `threads` gespeichert, und auch gleich gestartet.

Jetzt muss das `main`-Programm warten, bis alle Threads fertig sind. Dafür hat ein Thread die Methode `join()`. Sie unterbricht das `main`-Programm bis er fertig ist. Es ist wichtig zu wissen, dass ein Thread, der fertig ist, d.h. die `run`-Methode ist am Ende, immer noch als Datenstruktur existiert. Alle seine Instanzvariablen sind noch abrufbar.

Nachdem alle Threads fertig sind, müssen die Teilergebnisse noch miteinander multipliziert werden. Die Teilergebnisse kann man sich mit `threads[i].fakultaet` abrufen.

```java
import java.math.BigInteger;

public class Fakultaet extends Thread {
    public static int n;
    public int start;
    public int schritte;
    public BigInteger fakultaet = BigInteger.ONE;

    public Fakultaet(int start, int schritte) { // Der Konstruktor
        this.start = start;
        this.schritte = schritte;}

    public void run() { // Multiplikation mit Schrittweite
        for(int i = start; i <= n; i += schritte) {
            fakultaet = fakultaet.multiply(BigInteger.valueOf(i));}}


    public static void main(String[] args) {
        n = 500000;
        int nThreads = Runtime.getRuntime().availableProcessors();
        Fakultaet[] threads = new Fakultaet[nThreads];
        for(int i = 0; i < nThreads; i++) {
            threads[i] = new Fakultaet(i+1,nThreads);
            threads[i].start();}
        try{
            for(int i = 0; i < nThreads; i++) {
                threads[i].join();}
        } catch(InterruptedException e) {} // muss leider sein

        BigInteger ergebnis = BigInteger.ONE;
        for(int i = 0; i < nThreads; i++) {
            ergebnis = ergebnis.multiply(threads[i].fakultaet);}
        System.out.println(ergebnis);
        }
}
```

Ob man als Anzahl der Threads genau die Anzahl der Prozessoren nimmt, oder noch mehr, muss man ausprobieren. Man kann sich die CPU-Belastung ansehen. Wenn diese bei 100% ist, lohnt es sich nicht, noch mehr Threads zu verwenden.

Hier sind ein paar Ergebnisse für verschiedene n. $|n!|$ ist die Anzahl von Dezimalstellen im Ergebnis. Der verwendete Rechner hat eine Intel Core i7-6700 CPU, 2.6 GHerz, mit 8 logischen Prozessoren (Rechenkernen). Es wurde zunächst seriell, dann einmal mit 8 Threads und einmal mit 16 Threads gearbeitet. Die Zahlen geben die Rechenzeit in Millisekunden an.

n	$\|n!\|$	seriell	parallel(8)	parallel(16)
10.000	35.660	62	46	63
100.000	456.574	2.641	594	704
500.000	2.632.342	69.865	5.807	5.423
1.000.000	5.565.709	306.460	23.849	17.599

Die Parallelisierung bringt offensichtlich erst etwas bei wirklich riesigen Zahlen. Dann hat sie aber den erwarteten Beschleunigungseffekt. Mehr Threads als Prozessoren zu verwenden ist bei kleineren Zahlen eher kontraproduktiv. Bei ganz riesigen Zahlen hilft es etwas.

3.2 Beispiel: Paralleles Bubblesort

Das nächste Beispiel illustriert, dass auch bei einem einfachen Algorithmus schon echte Kreativität nötig ist, um ihn zu parallelisieren.

Bubblesort funktioniert so, dass immer benachbarte Paare verglichen werden. Falls ihre Reihenfolge falsch ist, wird sie umgedreht. In einem ersten Durchgang vergleicht man Objekt 1 mit 2, dann 2 mit 3, usw. Diese Durchgänge wiederholt man so lange, bis es keine Änderung mehr gibt.

Wie kann man das parallelisieren?

Die folgende Möglichkeit ist am besten geeignet wenn man ungefähr so viele Prozessoren hat, wie die Liste, die man sortieren möchte, lang ist.

Der Algorithmus unterscheidet 2 Phasen, die ungerade Phase und die gerade Phase. In beiden Phasen werden wiederum Nachbarelement verglichen, *aber alle parallel*!

In der ungeraden Phase wird verglichen: Element 1 mit 2, 3 mit 4, 5 mit 6 usw., jedes Paar parallel in einem separaten Thread.
In der geraden Phase wird verglichen: Element 2 mit 3, 4 mit 5, 5 mit 6 usw., auch alle parallel in separaten Threads.

Das macht man wiederum so lange, bis es keine Änderung mehr gibt.

Im folgenden Beispiel wird eine Liste von Zahlen damit sortiert. In den einzelnen Zeilen werden alle Paare (durch | getrennt) parallel verglichen und evtl. vertauscht.

Phase	Elemente		
	5 3 7 6 10 1 2 8		
ungerade	5 3 | 7 6 | 10 1 | 2 8	gerade	3 | 5 1 | 6 2 | 7 8 | 10
	3 5 | 6 7 | 1 10 | 2 8		3 | 1 5 | 2 6 | 7 8 | 10
gerade	3 | 5 6 | 7 1 | 10 2 | 8	ungerade	3 1 | 5 2 | 6 7 | 8 10
	3 | 5 6 | 1 7 | 2 10 | 8		1 3 | 2 5 | 6 7 | 8 10
ungerade	3 5 | 6 1 | 7 2 | 10 8	gerade	1 | 3 2 | 5 6 | 7 8 | 10
	3 5 | 1 6 | 2 7 | 8 10		1 | 2 3 | 5 6 | 7 8 | 10

Das Verfahren braucht im schlimmsten Fall so viele Schritte, wie die Liste lang ist. In diesem Fall muss ein Element durch die ganze Liste wandern. Ob das wirklich ein praktikables Verfahren ist, hängt allerdings sehr von der Hardware des Rechners ab.

3.3 Serverprogrammierung

Eine sehr typische Anwendung von Threads ist die Serverprogrammierung. Ein Server soll *parallel* mehrere Clients bedienen. Daher muss für jeden neuen Client ein eigener Thread gestartet werden, der genau diesen Client bedient.

Als Beispiel implementieren wir einen Server, der einfach nur die Fakultätsfunktion berechnet und das Ergebnis an den Client zurück gibt.

Zunächst implementieren wir die reine Serverklasse. Sie öffnet einen `ServerSocket` an einem gegebenen Port und geht dann in eine Endlosschleife. Dort wartet sie mit `sever.accept()` auf einen Client. Sobald sich ein Client verbindet, wird mit `new Fakultaet(client)` ein Thread erzeugt, und gleich gestartet. Das main-Programm erzeugt den Server mit port = 3141 und startet ihn dann.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {
  private int port;
  public Server(int port) {this.port = port;}

  private void startServing() throws IOException{
    ServerSocket server = new ServerSocket( port );
    while (true){
        Socket client = server.accept();
        new Fakultaet(client).start();}}

  public static void main( String[] args ) throws IOException{
    Server server = new Server(3141);
    server.startServing();}
  }
```

Die eigentliche Berechnung geschieht in dem Thread `Fakultaet`. Die run-Methode holt sich die Input- und Output-Streams des client-Sockets, liest die Zahl n (etwas umständlich über ein Byte-Array), berechnet $n!$, und schreibt das Ergebnis als Byte-Array in den OutputStream.

```
import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.math.BigInteger;
```

```
import java.net.Socket;
import java.nio.ByteBuffer;



public class Fakultaet extends Thread {
    Socket client;
    public Fakultaet(Socket client) {this.client = client;}

    public void run() {
        try{
            InputStream  in  = client.getInputStream();
            OutputStream out = client.getOutputStream();
            byte[] b = new byte[4];
            in.read(b);
            int n = ByteBuffer.wrap(b).getInt();
            BigInteger fakultaet = BigInteger.ONE;
            for(int i = 2; i <= n; ++i) {
                fakultaet = fakultaet.multiply(BigInteger.valueOf(i));}
            byte[] resultat = fakultaet.toByteArray();
            out.write(resultat.length);
            out.write(resultat);
            client.close();}
        catch(IOException e) {}
    }
}
```

Zum Testen brauchen wir noch eine Client-Klasse. In deren main-Methode wird zunächst mit `new Socket( "localhost", 3141 )` eine Verbindung zum Server aufgebaut. Vom Socket kann man sich die Input- und OutputStreams holen. Jetzt wird die Zahl n (im Beispiel $n = 10000$) als Byte-Array in den OutputStream geschrieben, und das Ergebnis ebenfalls als ByteArray eingelesen, und dann als BigInteger ausgedruckt.

```
import java.net.*;
import java.io.*;
import java.math.BigInteger;
import java.nio.ByteBuffer;

public class Client {
  public static void main( String[] args )
  {
    int n = 10000;
    try
    {
      Socket server = new Socket( "localhost", 3141 );
      InputStream   in = server.getInputStream();
      OutputStream out = server.getOutputStream();
      out.write(ByteBuffer.allocate(4).putInt(n).array());
```

```java
      int laenge = in.read();
      byte[] resultat = new byte[laenge];
      in.read(resultat);
      System.out.println(new BigInteger(resultat).toString());
    }
    catch ( UnknownHostException e ) {
      e.printStackTrace();
    }
    catch ( IOException e ) {
      e.printStackTrace();
    }
  }
}
```

Die Fehlerbehandlung in diesem Beispiel ist sehr rudimentär. Für echte Anwendungen muss man das natürlich noch viel besser machen.

4 Race Conditions in Java

Die Parallelisierung mit Threads hat die gleichen Probleme wie die Parallelisierung mit Prozessen: Deadlocks und Race Conditions. Deadlocks sind gegenseitige Verklemmungen: Thread 1 hält Ressource 1 und will Ressource 2. Thread 2 hält Ressource 2 und will Ressource 1. Keiner kann weiter machen. Gegen dieses Problem hilft nur eine entsprechende Organisation der Programme, so dass diese Situation nicht auftritt. Programmiersprachen selbst bieten dagegen kaum Hilfsmittel.

Race Conditions treten auf, wenn ein Thread sich eine lokale Kopie einer Ressource macht, damit rechnet, und dann die evtl. veränderte Kopie wieder zurück speichert. In der Zwischenzeit könnte ein anderer Thread das Original verändert haben, ohne dass der erste Thread das gemerkt hat.

Das kann insbesondere passieren, wenn Daten aus Files oder Datenbanken in den Arbeitsspeicher kopiert werden, oder wenn Daten aus dem Hauptspeicher in das Cache oder die Register kopiert werden.

Java stellt verschiedene Hilfsmittel zur Verfügung, damit das nicht passiert.

4.1 Monitore

Programmabschnitte, in denen Race Conditions passieren können, nennt man *kritische Bereiche*. Die beste Art, diese zu verhindern, ist, dafür zu sorgen, dass jeweils immer nur ein Thread in einen kritischen Bereich kommen kann. Wenn z.B. Daten aus einer Datenbank geladen werden, verändert werden und wieder zurück geschrieben werden, dann muss man verhindern, dass das gleichzeitig zwei Threads machen. Eine Möglichkeit dafür ist, den Codebereich mit einer Sperre, einem sog. *Monitor* zu versehen. Der Monitor sorgt dafür, dass immer nur ein Thread den Codebereich durchlaufen kann.

In Java gibt es dafür das Schlüsselwort *synchronized*. Wir verdeutlichen seinen Einsatz und seine Wirkung an einem kleinen Programm, welches Bankkonten bearbeitet. Das Beispiel ist nicht lauffähig, weil es keine Anbindung an eine Datenbank mit Bankkonten gibt. Es soll nur die Wirkung von synchronized verdeutlichen.

```
public class Konto {
  public synchronized void einzahlen(int betrag){
    int kontostand = ladeKontostand();
    kontostand += betrag;
    sichereKontostand(kontostand);}

  public synchronized void abheben(int betrag){
    int kontostand = ladeKontostand();
    kontostand -= betrag;
    sichereKontostand(kontostand);}
}
```

Jetzt definieren wir eine Thread-Klasse Ueberweisung, deren run-Methode einen Betrag einzahlt oder abhebt. Instanzen von so einer Klasse könnten z.B. erzeugt werden, wenn ein Bankautomat mit der Bank Kontakt aufnimmt.

```
public class Ueberweisung extends Thread {
  private Konto konto;
  int betrag betrag;

  public Ueberweisung(Konto konto, int betrag) {
          this.konto = konto; this.betrag = betrag;}

  public void run() {
    if(betrag > 0) {konto.einzahlen(betrag);}
    else {konto.abheben(-betrag);}
}
```

Schließlich brauchen wir ein paar Überweisungen.

```
public class Test {
  public static void main(String[] args) {
    Konto mayerKonto = new Konto();
    Ueberweisung ehemann = new Ueberweisung(mayerkonto,500);
    Ueberweisung ehefrau = new Ueberweisung(mayerkonto,-300);
        ehemann.start();
        ehefrau.start();
}
```

Es werden jetzt zwei Threads erzeugt und mit `start()` gestartet.

Die Anweisung `ehemann.start()` aktiviert die run-Methode des ersten Threads. Diese wiederum ruft `mayerkonto.einzahlen(500);` auf. In diesem Moment wirkt *synchronized*: da die Metho-

de `einzahlen` vom `mayerkonto` aufgerufen wurde, etabliert sich das Objekt `mayerkonto` als Monitor und sperrt alle weiteren Zugriffe auf synchronized Methoden, die für es aufgerufen werden können, in diesem Fall auf beide Methoden, einzahlen und abheben.

Wenn jetzt gleichzeitig der Thread `ehefrau` die run-Methode aufruft und dort `mayerkonto.abheben(300);` aufrufen will, verhindert das der Monitor `mayerkonto`. Erst wenn der erste Thread fertig ist, gibt er die Methode frei.

Gäbe es ein weiteres Konto, z.B. `muellerkonto`, und dieses will auch etwas einzahlen, dann hätte der Monitor `mayerkonto` nichts dagegen. Das dürfte parallel geschehen.

```
Konto muellerKonto = new Konto();
Ueberweisung mueller = new Ueberweisung(muellerkonto,500);
```

Zusammenfassend: Wirkung von `synchronized` bei Methoden:
Wird eine mit `synchronized` markierte Methode `m(...)` von einem Objekt `o` aufgerufen, `o.m(...)` dann etabliert sich `o` als Monitor. Wird dann `m(...)` selbst oder irgendeine andere `synchronized` Methode von dem Objekt `o` aufgerufen (das kann dann nur von einem anderen Thread aus sein), dann blockiert der Monitor den Aufruf so lange bis der erste Aufruf `o.m(...)` fertig ist.

Das Schlüsselwort `synchronized` vor einer Methode bewirkt eine Blockade der ganzen Methode. Will man stattdessen nur einen Teil der Methode blockieren, dann kann man `synchronized` im Code verwenden. Die Syntax dazu ist `synchronized(monitor) {...}` Der `monitor` muss natürlich ein geeignetes Objekt sein, z.B. das betroffene Konto, also this, wie im obigen Beispiel.

4.2 volatile

Es gibt noch eine viel heimtückischere Fehlerquelle für Race Conditions. Die einzelnen Threads können `static` Variablen in das Cache laden, und dort verändern, bevor sie wieder in den Arbeitsspeicher zurück kopiert werden. Machen das zwei Threads gleichzeitig mit *derselben* Variablen, dann gibt es zwei unterschiedliche Kopien im Cache. Werden die wieder zurück in den Arbeitsspeicher kopiert, dann überschreibt die zweite einfach den Wert der ersten. Um das zu verhindern, markiert man `static` Variablen, die von mehreren Threads benutzt werden, mit dem Schlüsselwort `volatile`. Das sorgt dafür, dass keine Kopie im Cache erzeugt wird.

Allerdings ist ein direkter Zugriff von mehreren Threads auf dieselbe Variable nicht zu empfehlen. Eine sauberere Art der Programmierung wäre den Zugriff über synchronisierte setter- und getter-Methoden laufen zu lassen.

`synchronized` und `volatile` erzeugen einen Mehraufwand bei der Verarbeitung. Daher sollte man immer genau analysieren, wo das notwendig ist, und wo nicht.

5 Zusammenfassung

Thread-Programmierung ist ein sehr elegantes und effizientes Hilfsmittel, um Programme zu parallelisieren. Für mache Anwendungen sind sie absolut notwendig, z.B. um Server zu programmieren, die mehrere Clients gleichzeitig bedienen sollen. In anderen Anwendungen, insbesondere der Parallelisierung von Algorithmen, kann sie eine deutliche Effizienzsteigerung bringen, muss aber nicht. Gerade bei der Parallelisierung von Algorithmen sollte man sehr genau deren Verhalten analysieren. Wenn die Verwaltung der Threads mehr Ressourcen frisst, als die Parallelisierung bringt, nutzt sie nichts.

Die Programmiersprache Java unterstützt die Thread-Programmierung in vielfältiger Weise, nicht nur mit den oben beschriebenen Techniken. Es lohnt sich sicher, die entsprechenden Dokumentationen anzusehen.

Stichwortverzeichnis

Deadlock, 12

Kernel-Level Threads, 4
kritischer Bereich, 12

Monitor, 12

Race Conditions, 12

synchronized, 13

Thread Pool, 5
Threads, 2

User-Level Threads, 4

volatile, 14