Nebenläufige Programmierung I

Martin Wirsing

in Zusammenarbeit mit
Matthias Hölzl, Piotr Kosiuczenko, Dirk Pattinson

07/03

Ziele

- Grundlegende Begriffe der nebenläufigen Programmierung verstehen lernen
- Nebenläufige Programme in Java schreiben lernen

Nebenläufige und verteilte Systeme

- Ein **System** ist von seiner Umgebung abgegrenzte Anordnung von Komponenten.
- Können die Aktivitäten gleichzeitig stattfinden, spricht man von einem **parallel ablaufenden (nebenläufigen) System**.
- Ist das System aus räumlich verteilten Komponenten aufgebaut, spricht man von einem **verteilten System**.
- Programme, wie wir sie bisher geschrieben haben, arbeiten sequentiell.

Beispiel: Banktransaktion „single-threaded"

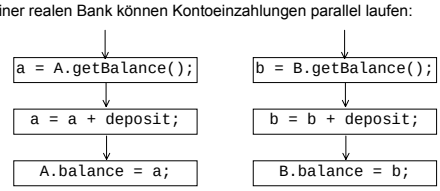
Der Stand eines Konto wird abgefragt, eine Summe „deposit" eingezahlt, die Kontostandartvariable um einen Betrag „deposit" erhöht und der neue Wert als neuer Kontostand angetragen:

```
a = A.getBalance();      // erfragte Kontostand a von Konte A
        ↓
a = a + deposit;         // erhöhe a
        ↓
A.balance = a;           // setze Kontostand auf neuen Wert von a
```

Bemerkung: Auch reale Bankautomaten und Computerprogramme laufen in solchen Sequenzen ab, die man „Thread" (Kontrollfluß) nennt. Das obige Prigramm ist ein „**single-threaded**" Programm.

Beispiel: Banktransaktion „single-threaded"

In einer realen Bank können Kontoeinzahlungen parallel laufen:

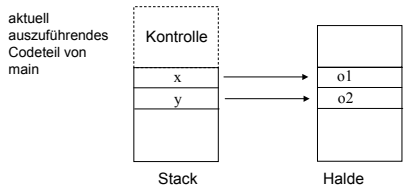


Bemerkung: Innerhalb eines Computers wird dies „multi-threading" genannt. Ein Thread kann eine Aufgabe unabhängig von anderen Threads erfüllen. Ebenso wie zwei Bankautomaten die gleichen Konten benutzen können, können Threads Objekte gemeinsam benutzen.

Thread

- Ein **Thread** ist ein Teil eines Programms, das unabhängig von von anderen Teilen des Programms ausgeführt werden kann, Es repräsentiert eine einzige Sequenz von Anweisungen, d.h. einen sequentiellen Kontrollfluss, der nebenläufig zu anderen Threads ausgeführt werden kann, und der Daten gemeinsam mit anderen Threads benutzt.
- Aus Betriebssystemsicht wird ein **Prozess** verstanden als eine **abstrakte Maschine, die eine Berechnung ausführt**. Da ein Thread im Allgemeinen zusammen mit anderen Threads auf der gleichen abstrakten Maschine läuft und Ressourcen mit anderen Threads teilt, ist ein Thread ein „**leichtgewichtiger Prozess**".
- Ein nebenläufiges Java-Programm besteht aus mehreren Threads, die über **gemeinsame Objekte** miteinander kommunizieren.

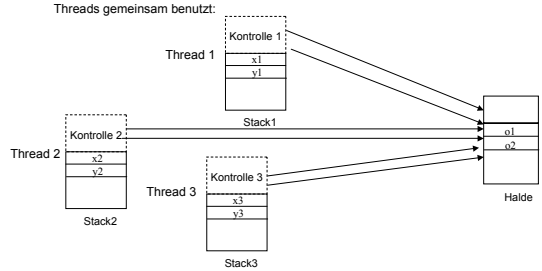
Sequentielles Speichermodell von Java



Hier entspricht die Ausführung von main einem (single-)Thread.

Speichermodell für nebenläufige Programme

Ein Thread ist ein „leichtgewichtiger" Prozeß, der die Halde mit anderen Threads gemeinsam benutzt:



Variablen in nebenläufigen Prozessen

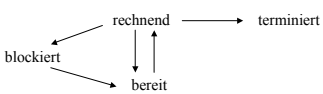
- **Lokale Variable:**
  - Lokale Variablen sind lokal für jeden Thread;
  - Änderungen lokaler Variablen haben keinen Einfluß auf andere Threads.
- **Instanzvariable:**
  - Objekte residieren in einem globalen Speicher.
  - Änderungen von Instanzvariablen können die Werte anderer Threads beeinflussen.

Scheduling

- Wenn ein Programm mehrere Threads enthält, wird ihre Ausführung durch das Java-Laufzeitsystem geregelt.
- Der „Scheduler" unter Windows NT implementiert eine „Zeitschrittstrategie", bei der jeder Thread eine gewisse Prozessorzeit zugeteilt bekommt und dann unterbrochen und in einer Warteschlange hintenangestellt wird.
- Ältere Implementierungen haben ein solches Verfahren nicht. Ein arbeitender Thread wird nicht unterbrochen und behält den Prozessor, bis er terminiert oder sich selbst in einen Wartezustand bringt.
- Die Strategien der Implementierungen können also verschieden sein. Deshalb darf man sich im Programm **nicht** auf ein bestimmtes Verfahren verlassen.

Zustände eines Threads

rechnend:	Der Thread wird im Prozessor ausgeführt.
blockiert:	Der Thread braucht ein Betriebsmittel, das temporär nicht vorhanden ist
bereit:	Der Thread ist rechenbereit, hat aber nicht Zugriff auf den Prozessor.
terminiert:	Der Thread ist beendet.



Das verfeinerte Nebenläufigkeitsmodell

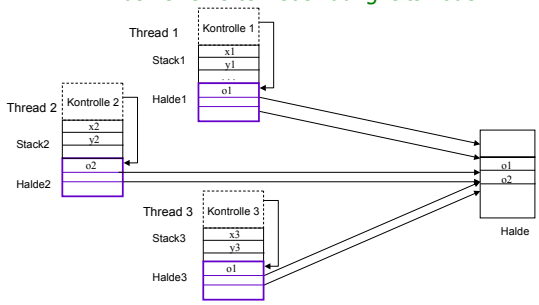
In Wirklichkeit besitzt jeder Thread auch eine lokale Halde. Nach genau festgelegten Regeln werden Daten

zwischen **der lokalen Kontrolle und der lokalen Halde** sowie

zwischen **der lokalen und der globalen Halde**

ausgetauscht (Forderung nach schwacher Konsistenz.)

Das verfeinerte Nebenläufigkeitsmodell



Threads in Java

- Ein **Thread in Java** wird repräsentiert durch eine **Instanz der Klasse Thread** (oder einer Subklasse von Thread)
- Ein nebenläufiges Java-Programm besteht aus mehreren Threads, die über **gemeinsame Objekte** miteinander kommunizieren.

Die Klasse java.lang.Thread

Konstruktoren:

```
Thread ()                 //erzeugt einen neuen Thread
Thread (Runnable r)       //erzeugt einen neuen Thread, der die
                          //run-Methode von r ausführt
```

Methoden:

```
run ()                    // determines the flow of control
start ()                  // startet einen Thread und ruft run auf
static sleep(long s)      // unterbricht den gerade ausgeführten Thread um s msec
static yield ()
. . .
```


Erzeugung von Threads

2 Techniken:

- als Objekt der Klasse `Thread` oder einer (Sub-) Klasse davon
- durch Implementierung der Schnittstelle `Runnable`

Erzeugen von Threads in Erben von Thread

Schema:

```
class C extends Tread
{...
   public void run
   {<<bestimmt den Kontrollfluß des Thread>>
   }
   public static void main (String[] args)
   { C t = new C();
     t.start(); ...
   }
```

ruft run auf

Beispiel: PingPong

```
public class PingPong extends Thread
{       private String word;          // das zu druckende Wort
        private int delay;            // der Unterbrechungszeitraum

        public PingPong(String whatToSay, int delayTime)
        {       word = whatToSay;
                delay = delayTime;
        }
        public void run()
        {       try
                { for (;;)            // unendliche Schleife
                        {       System.out.println(word + " ");
                                sleep(delay); //auf das n"achste Mal warten
                        }
                }
                catch(InterruptedException e)
                        { return ; // beende diesen Thread
                }
        }
```

Beispiel: PingPong

```
public static void main(String[] args)
{       new PingPong("ping", 33).start();     // 1/30 sec.
        new PingPong("PONG",100).start();     // 1/10 sec.
}
}
```

stoppt für 30 ms

stoppt für 100 ms

Bemerkung:

- Die `run`-Methode von Ping-Pong terminiert nicht und muß deshalb explizit abgebrochen werden mit der `InterruptedException` von `sleep`
- Mögliche Ausgabe

  `ping PONG ping ping PONG ping ping ping PONG...`

Erzeugen von Threads mit `Runnable`

- Eine Subklasse von `Thread` kann nur von `Thread` (oder einer Subklasse davon) erben.
- Mit Hilfe der Schnittstelle `Runnable` kann ein Thread auch von anderen Klassen erben:

```
public interface Runnable
{   void run();
}
```

- Mit den Konstruktoren (der Klasse Thread)

```
public Thread (Runnable r)
public Thread (Runnable r, String name)
```

konstruiert man einen neuen Thread, der die `run`-Methode von `r` verwendet, `name` gibt einen Namen für den Thread an.

Schematisches Aufbau eines Runnable Threads

Schema:

```
class ConcreteRunnable implements Runnable {
        ...
        public void run()
        {               //Code für den Kontrollfluß
        }
        public static void main (...)
        {   Runnable r = new ConcreteRunnable(...);
                                  //neues Runnable-Objekt
            Thread t = new Thread (r); // neues Thread-Objekt
            t.start();                 // Aufruf von r.run();
            }
```

oder `ConcreteRunnable`

Beispiel: PingPong

```
public class RunPingPong extends Thread
{       << wie vorher  >>
        public static void main(String[] args)
        {       Runnable ping = new RunPingPong("ping", 33);
                RunPingPong pong = new RunPingPong("PONG",100);
                new Thread(ping).start();
                new Thread(pong).start();
        }
}
```

Bemerkung:

- Die Klasse `Thread` und damit auch die Erbenklasse `RunPingPong` implementieren die Schnittstelle `Runnable`. Deshalb ist das `main`-Programm (syntaktisch) korrekt.

Probleme bei nebenläufiger Programmierung

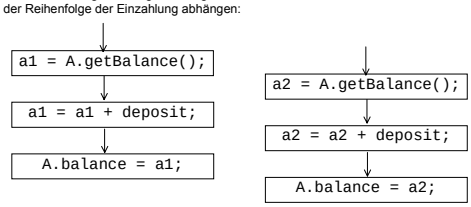
Beispiel:

Das folgende (unschöne) Programm für `deposit` spiegelt das Verhalten eines Bankangestellten wider:

```
void strangeDeposit(double d)
{
   double hilf = getBalance();
   hilf = hilf + d;
   balance = hilf;
}
```


Probleme bei nebenläufiger Programmierung

Wenn zwei Kunden gleichzeitig auf das gleiche Konto einzahlen wollen, kann der Kontostand von der Reihenfolge der Einzahlung abhängen:



Jeder Bankangestellte geht zum Kontobuch, sieht den Wert nach und geht dann später wieder hin, um den neuen Wert einzutragen. Damit ist der erste Eintrag verloren. ⇒ „Race Condition", Interferenz zwischen Threads

Race Conditions

Jeder Bankangestellte geht zum Kontobuch, sieht den Wert nach und geht dann später wieder hin, um den neuen Wert einzutragen. Damit ist der erste Eintrag verloren.

⇒ Man erhält eine
„Race Condition“, d.h. eine Interferenz zwischen Threads

Um „Race Conditions“ zu vermeiden, hat man früher eine Bemerkung auf das Konto A geschrieben: „Ich arbeite an dem Konto“.

In Java setzt man dafür eine Sperre („Lock“) auf ein Objekt. Eine derartige Sperre ist mit dem Objekt assoziiert, um bestimmen zu können, ob es benutzt ist.

Sicherheit durch Synchronisation

- Interaktion zwischen Threads kann zu unerwünschtem Verhalten führen.
- Man unterscheidet zwei Arten von Eigenschaften nebenläufiger Programme:
  - Sicherheit:
    - Eigenschaft, daß nie etwas Unerwünschtes geschieht
  - Lebendigkeit:
    - Eigenschaft, dass immer irgendetwas geschieht
- Sicherheitsfehler führen zu unerwünschtem Laufzeitverhalten
- Lebendigkeitsprobleme führen dazu, daß das Programm anhält, obwohl noch nicht alle Threads terminiert haben.

Sicherheit und Nebenläufigkeit

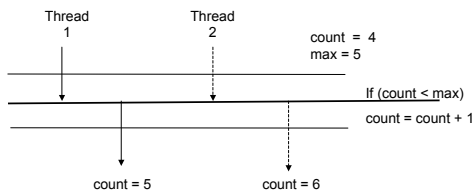
- Sicherheitsprobleme basieren häufig auf
  - **Lese / Schreibkonflikten**

    Der Zugriff eines lesenden Kunden auf ein Objekt während eines noch nicht beendeten Schreibvorgangs kann zum lesen inkorrekter Daten führen

    Beispiel: Lesen einer Reihung während alle Feldelemente verdoppelt werden
  - **Schreib/Schreibkonflikte**

    Inkonsistente Zuweisung an Variablen durch nebenläufiges Ausführen von zustandsändernden Methoden (siehe Bankangestellten-Beispiel)

Beispiel: Schreib/Schreib-Konflikt (ohne Synchronisation)



Beispiel: Schreib/Schreib-Konflikt (ohne Synchronisation) - Ablauf

- Die lokalen Variablen haben die Werte: count = 4, count = 5
- Thread 1 führt den ersten Teil der Anweisung aus

  if (4 < 5) (also count = max)
- Threads 2 führt den ersten Teil der Anweisung aus

  if (4 < 5) (also count = max)
- Thread 1 führt den 2. Teil der Anweisung aus

  Ergebnis: 5 = 4 + 1 (count ++)
- Thread 2 führt den 2. Teil der Anweisung aus

  Ergebnis: 6 = 5 + 1 (count++)

Synchronisation von Methoden

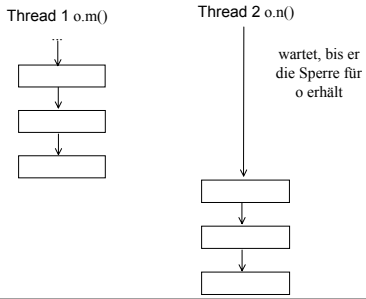
Vermeidung von „Race Conditions“ durch Synchronisation:

synchronized T m(T_1x)

Nicht synchronisierte Methoden können auf gesperrte Objekte zugreifen

- Wenn ein Thread `o.m(a)` aufruft, wird eine Sperre auf o gesetzt.
- Jeder andere Thread, das o mit einer synchronized-Methode aufruft, wird blockiert, bis die Sperre (nach Beendigung der Ausführung von `o.m(a)`) aufgehoben wird.
- Durch Synchronisation erreicht man folgende **Sicherheitseigenschaft**: Es können niemals zwei Threads gleichzeitig auf einem gemeinsamen Datenbereich zugreifen.

Beispiel: Wirkung der Synchronisation



Bemerkungen

- **Wechselseitiger Ausschluß:**

  Man spricht von wechselseitigem Ausschluß, wenn zu jedem Zeitpunkt nie mehr als ein Prozess auf ein Objekt zugreifen kann.
- Statische Methoden können auch synchronisiert werden
- Wird eine synchronisierte Methode in einem Erben überschrieben, so kann sie, muß aber nicht synchronisiert sein.

Beispiel: BankAccount

Die folgende Account-Klasse synchronisiert die deposit-Operation und garantiert so, dass während der Ausführung von deposit keine andere synchronisierte Operation auf das aktuelle Konto zugreifen kann.

```
public class Account
{ ...
   public synchronized double getBalance()
   { ... }
   public synchronized void deposit1(double d)
   {   int hilf = getBalance();
       hilf = hilf + d;
       balance = hilf;
   }
}
```


Zusammenfassung

- Java unterstützt Nebenläufigkeit durch „leichtgewichtige“ Prozesse, sogenannte Threads, die über die gemeinsame Halde und damit über gemeinsam benutzte Objekte miteinander kommunizieren.
- Nebenläufigkeit wirft Sicherheits- und Lebendigkeitsprobleme auf. Gemeinsam benutzte Objekte müssen synchronisiert werden.