Kapitel 12 - Nebenläufigkeit: Arbeiten mit Threads

Inhalt - Überblick

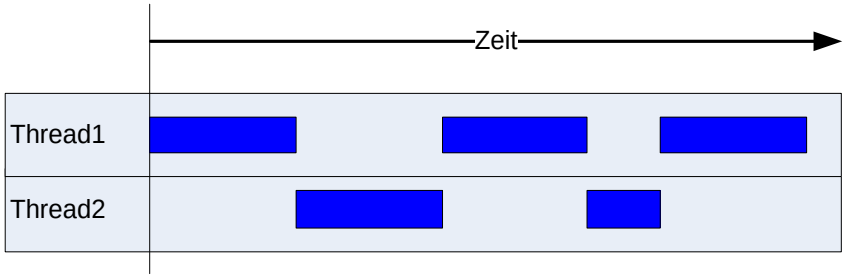
1. **Java Grundlagen: Entwicklungszyklus, Entwicklungsumgebung**
2. Datentypen, Kodierung, Binärzahlen, Variablen, Arrays
3. Ausdrücke, Operatoren, Schleifen und Verzweigungen
4. Blöcke, Sichtbarkeit und Methoden (Teil 1)
5. Grundkonzepte der Objektorientierung
6. Objektorientierung: Sichtbarkeit, Vererbung, Methoden (Teil 2), Konstruktor
7. Packages, lokale Klassen, abstrakte Klassen und Methoden, Interfaces, enum
8. Arbeiten mit Objekten: Identität, Listen, Komparatoren, Kopien, Wrapper, Iterator
9. Fehlerbehandlung: Exceptions und Logging
10. Utilities: Math, Date, Calendar, System, Random
11. Rekursion, Sortieralgorithmen und Collections
12. Nebenläufigkeit: Arbeiten mit Threads
13. Benutzeroberflächen mit Swing
14. Streams: Auf Dateien und auf das Netzwerk zugreifen

Was ist ein Prozess?

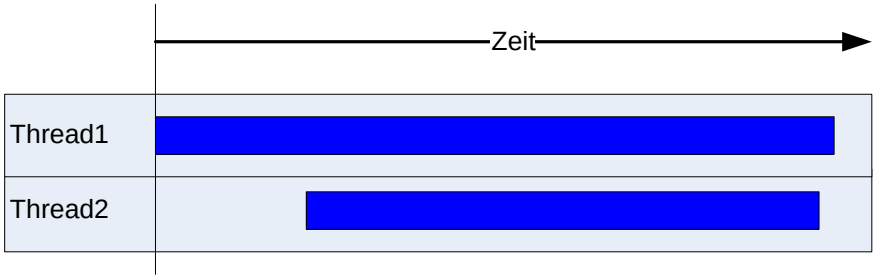
- Ein Prozess ist ein spezieller Ablauf.
- Ein Betriebsystem verwaltet Prozesse anhand eines Prozessleitblocks (PLB, Task Control Block TCB)
- Der Prozess „besitzt“
- einen Adressraum mit
  - einem eigenen Stack
  - eigene Daten
  - Zugeordneter Programmcode
- eine Registermenge inkl. eigenem Programmzähler
- Zustand
- geöffnete Dateien, Sockets, GUI, ...
- Zugriffsrechte
- Prozesse können parallel zueinander ablaufen, je nach Betriebssystem.
- → Alte Ausdrucksweise „Mehrprogrammbetrieb“

Was ist ein Thread?

- Ein Prozess kann mehrere Threads (Programmfäden, Kontrollfaden) enthalten.
- Man bezeichnet Threads auch als leichtgewichtige Prozesse (lightweight process)
- Threads laufen „quasi-parallel“ auf Einprozessorsystemen



- Threads können auf Mehrprozessorsystemen parallel laufen



Jeder Thread eines Prozesses besitzt exklusiv:

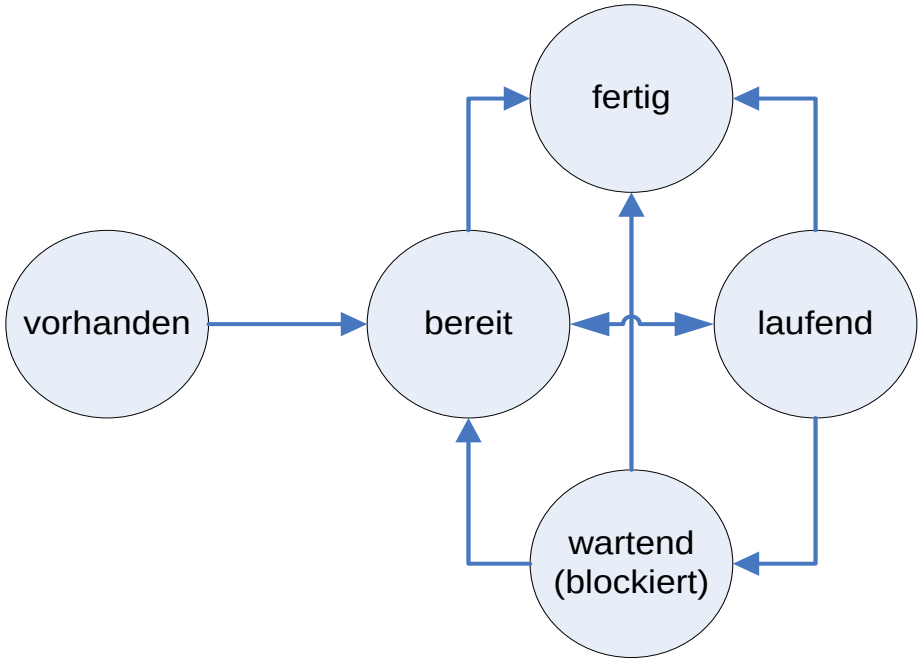
- einen eigenen Stack
- eine Registermenge incl. eigenem Programmzähler
- Zustand

Alle Threads eines Prozesses haben gemeinsam:

- den Adressraum
- globale Variable
- geöffnete Dateien, Sockets, GUI, ...
- Zugriffsrechte bei Start der Threads

Java: Threads eines Programms nutzen gemeinsam:

- static-Variablen
- ABER auch: Variablen auf Klassenebene in gemeinsam benutzten Objekten
- Nicht: Lokale Daten auf den Stacks der Threads



- **vorhanden**: das Thread Objekt wurde mit new erzeugt
- **bereit**: der Thread ist ablaufbereit, aber das Betriebssystem teilt die CPU (noch) nicht zu.
- **laufend**: der Thread „hat“ die CPU.
- **wartend**: dem Thread fehlt ein Betriebsmittel, er wurde bswp. Mit sleep() schlafen gelegt
- **fertig**: der Prozess ist beendet, d.h. die run Methode ist abgearbeitet

Zur Implementierung von Threads in Java gibt es die Klasse java.lang.Thread, die kurz als Thread bezeichnet wird.

Threads können in Java auf zwei verschiedene Arten definiert werden:

- als Unterklasse von Thread
- durch Implementierung der Runnable-Schnittstelle

Variante a: Wenn man in einem Programm die Klasse MeineKlasse mit der Haupteigenschaft „MeineKlasse ist ein Thread“ hat, sollte man MeineKlasse als Unterklasse von Thread implementieren.

Siehe: http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Variante b: Wenn man aber ein GUI-Objekt hat, das als Unterklasse irgendwelcher GUI-Klassen geschrieben ist, kann dies keine Unterklasse von Thread sein, denn Java kennt keine Mehrfachvererbung.

In diesem Fall müsste die Thread-Funktionalität durch Implementierung der Runnable-Schnittstelle implementiert werden, wobei die „Arbeit“ an einen Thread als Mitglied der Klasse delegiert wird.

```
class ErsterThread extends Thread {
  public void run () {
    for (int i = 0; i < 10; i++) {
      System.out.println (i + " ");
      try {
        sleep (Math.round (1000.0*Math.random ()));
      } catch (InterruptedException e) {
        System.out.println (e);
      }
    }
    System.out.println ("Ende Thread "+toString ());
  }
}
```

```
public class ErsterThreadDemo {
  static public void main (String args[]) {
    ErsterThread thread = new ErsterThread ();
    thread.start ();
    System.out.println ("Ende main");
    // Kein Warten auf das Ende des Threads
  }
}
```

```
class ZweiterThread implements Runnable {
  private Thread thread; // Der Thread zur Delegation

  public ZweiterThread () {
    thread = new Thread (this);
  }

  public void start () {
    thread.start ();
  }

  public void join () throws InterruptedException {
    thread.join ();
  }
```

```
  public void run () {
    for (int i = 0; i < 10; i++)
      try {
        thread.sleep (
        Math.round (1000.0*Math.random ()));
        System.out.println(thread.toString()+" "+i);
        // oder auch: thread.sleep (100);
      }
      catch (InterruptedException e) {
        System.out.println (e);
      }
  }
}
```

```
public class ZweiterThreadDemo {
  static public void main (String args[]) {
    ZweiterThread thread2 = new ZweiterThread ();
    thread2.start ();

    try {
      thread2.join ();
    } catch (InterruptedException e) {
      System.out.println (e);
    }
  }
}
```

- Gegeben ist ein Programm, welches einen Zähler x bearbeitet.
  - Ein Thread inkrementiert den Zähler n mal: x = x+1;
  - Ein Thread dekrementiert den Zähler n mal: x = x-1;
- Welchen Stand hat der Zähler am Ende?
- Vermutung: 0
- Tatsächlich ist jedes Ergebnis zwischen –n und +n möglich.
- Siehe Wettrennen.java

Konkurrierende Threads: Analyse des Problems

Thread1

```
   5:  getstatic #19;  // zähler  Stack
   8:  aload_0         // this  Stack
   9:  getfield #38;   // increment  Stack
  12:  ladd            // zähler+increment




  13:  putstatic #19;  // Wert -> zähler
Alle von Thread2 durchgeführten Änderungen
  sind überschrieben, also verloren!
```

Thread2

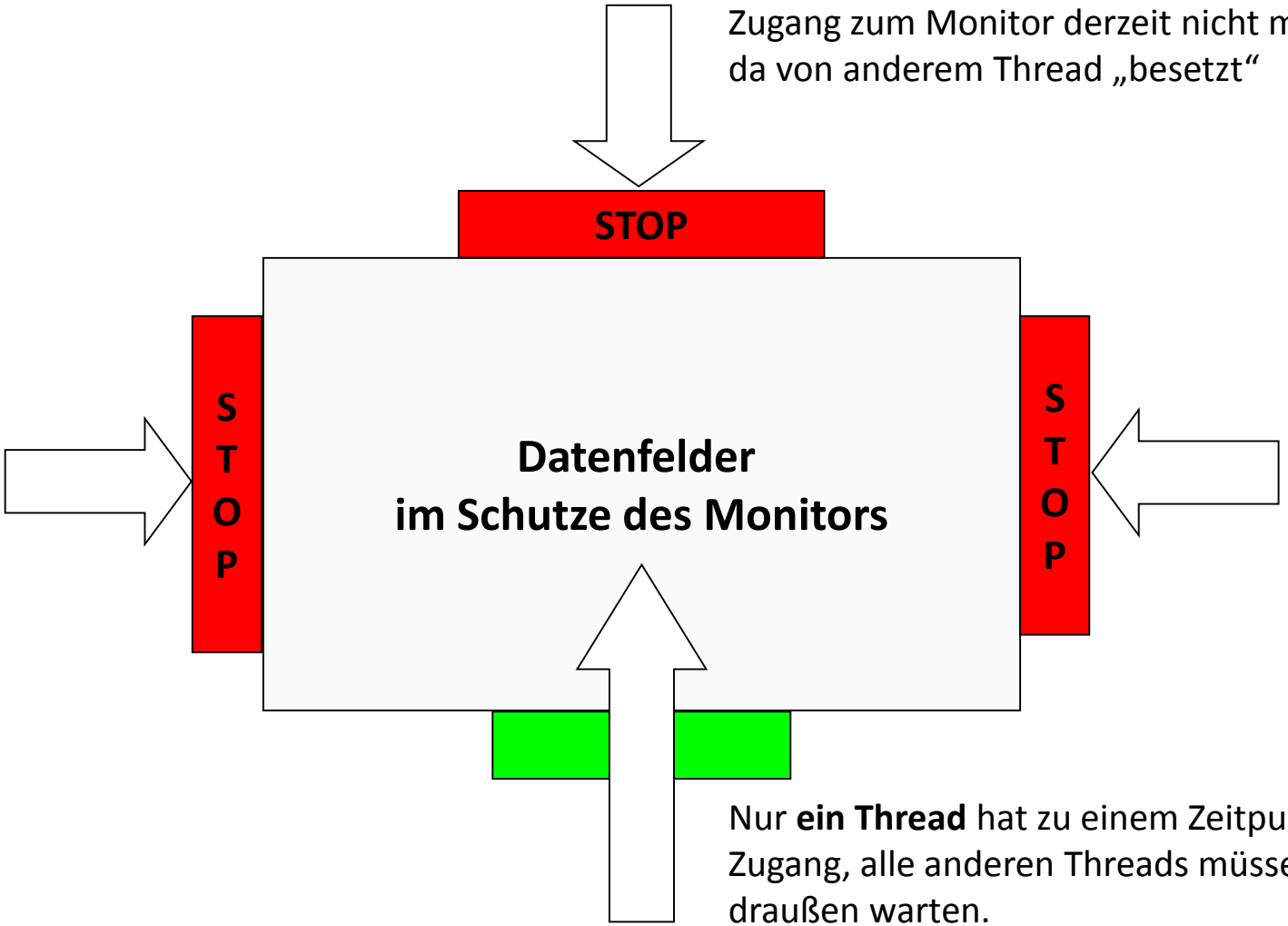
Unterbrechung

```
   5:  getstatic #19;  // zähler  Stack
   8:  aload_0         // this  Stack
   9:  getfield #38;   // increment  Stack
  12:  ladd            // zähler+increment
  13:  putstatic #19;  // Wert -> zähler

 … Wiederholen, bis das Betriebssystem Thread2
pausieren lässt
```

- Möglicher Ablauf
  - Thread1 liest den Inhalt von x in ein Register.
  - Thread2 liest den Inhalt von x in ein Register.
  - Thread2 erhöht „sein“ Register (1…n-mal).
  - Thread2 schreibt das Ergebnis zurück.
  - Thread1 erniedrigt „sein“ Register.
  - Thread1 schreibt das Ergebnis zurück.

Monitore: Koordination der Zugriffe



Zugriff auf Daten durch mehrere Threads ohne Koordination

```
class Punkt {
  private float x, y;
  … Konstruktoren
  void setzePunkt (float x, float y) {
    this.x = x;
    this.y = y;
  }

  Punkt liesPunkt () {
    return new Punkt (x, y);
  }
}
```

Durch eine Unterbrechung eines Threads, der gerade die Methode liesPunkt an der gekennzeichneten Stelle ausführt, durch einen anderen Thread, der die Methode setzePunkt ausführt, könnte der Fall eintreten, dass die Methode liesPunkt die x-Koordinate des alten Punktes und die y-Koordinate des neuen Punktes liefert.

Nicht ohne Koordination …

```
// Unsynchronisiert geht es schnell, aber.... falsch
  public void demoUnSync() {
    for (long i = 0; i < MAX; i++) {
      zähler = zähler + increment;
    }
  }
```

… sondern mit Koordination durch Monitor:

```
  // Dieses Objekt dient der Synchronisierung.
  // Deswegen nützt es nur dann, wenn es von allen
  // zu synchronisierenden Threads benutzt wird.
  static Object o      = new Object();
  // Synchronisierter Zugriff
  public void demoSync() {
    for (long i = 0; i < MAX; i++) {
      synchronized (o) {
        zähler = zähler + increment;
      }
    }
  }
```

Zugriff auf Daten durch mehrere Threads mit Koordination

```
class Punkt {
 private float x, y;
  … Konstruktoren …
  synchronized void setzePunkt (float x, float y) {
    this.x = x;
    this.y = y;
  }

  synchronized Punkt liesPunkt () {
    return new Punkt (x, y);
  }
}
```

Wenn die Zugriffsmethoden mit dem Schlüsselwort `synchronized` versehen sind, kann zu einem Zeitpunkt nur eine der `synchronized` -Methoden aufgerufen werden.

Falls derjenige Thread, der die Daten ausliest, unterbrochen wird und ein zweiter Thread mit Schreibzugriffen aktiv wird, muss der zweite Thread warten, bis der erste (der Leser) den Monitor verlassen hat.

- Das letzte Programm läuft korrekt, aber langsam
- Abhilfe ab Java 5 im Package java.util.concurrent:
- Variable mit atomaren (=unteilbaren) Operationen

```
static AtomicLong atomicZähler = new AtomicLong(0);

  // Ab JDK 1.5: Atomic - Operationen
  public void demoJava5() {
    for (long i = 0; i < MAX; i++) {
      atomicZähler.addAndGet(increment);
    }
  }
```

- Das Programm läuft korrekt und **dreimal so schnell** wie die Monitor-Variante!

Implementierung von langen Wartephasen

- Falls beispielsweise ein Thread zur Bildschirmanzeige von Daten auf diese Daten warten muss, die ein anderer Thread über das Internet heranholen muss, dann muss der erste Thread auf ein Ereignis vom zweiten Thread warten.
- Wenn der erste Thread dieses „Warten“ in Form von aktivem Warten dadurch implementiert, dass er in einer Schleife ständig eine Variable abfragt, wird unnötig CPU verbraucht, die für andere Aufgaben am Rechner fehlt.
- Außerdem könnte ein niederpriorer Thread daran gehindert werden, die Tätigkeit zu verrichten, auf deren Ende gewartet wird. Für dieses „Warten“ dient in Java der
- wait()-Aufruf
- Er sorgt dafür, dass dem aufrufenden Thread die CPU entzogen wird. Der Thread wird „schlafen gelegt“ und verbraucht danach keine CPU-Zeit mehr.
- Der Thread verlässt auf diesem Weg sofort den Monitor, sonst wäre dieser für den Zugang anderer Threads blockiert.

Benachrichtigung wartender Threads

- Wenn der zweite Thread die Daten geholt oder wenigstens einen brauchbaren Teil davon beschafft hat, muss der erste Thread wieder „aufgeweckt“ werden. Außerdem müssen die Daten übergeben werden. Dieses Problem wird in Java mit den Aufrufen
  notify (); *// Benachrichtige einen wartenden Thread*
  notifyAll (); *// Benachrichtige ALLE wartenden Threads*
  gelöst, die nur aus einem Monitor heraus absetzbar sind. Die Benachrichtigung gilt dann für Threads, die an eben diesem Monitor per wait() schlafen gelegt wurden.
- Der notify()-Aufruf befreit *einen* der schlafen gelegten Threads.
- Der Thread muss aber bei dem in Java implementierten Konzept der Monitore noch auf Zugang zum Monitor warten, denn der andere Thread verlässt infolge seines Aufrufs zur Benachrichtigung den Monitor keineswegs sofort.
- Außerdem könnte ein anderer Thread vor dem mit notify() aktivierten Thread Zugang zum Monitor erhalten.
- Siehe ConsumerProducer.java