Einstieg in die Programmierung von Computern – Teil IX

Im Teil 9 beschäftigen wir uns mit dem sehr wichtigen und anspruchsvollen Thema Multithreading bzw. Nebenläufigkeit in Java. Dabei geht es um die parallele Ausführung von Java-Code auf nur einem Mikroprozessor in einem Rechner. Die Programmiersprache Java ist dafür bestens ausgerüstet und bietet für Multithreading eine Menge Support.

Was ist ein Prozess?

Bislang haben wir in unserer Serie „Einstieg in die Programmierung von Computern mit Java“ alle Programme auf einem Mikroprozessor in einem sogenannten „Prozess“ zum Ablauf gebracht. Ein Prozess ist die Ausführung eines sequenziellen Programmes. Dabei setzt sich ein Prozess aus dem Programmcode und den dazugehörigen Daten zusammen und besitzt einen eigenen Adressraum. Dabei übernimmt die **virtuelle Speicherverwaltung** des Betriebssystems die Trennung der Adressräume der einzelnen Prozesse.

Die Steuerung eines Prozesses übernimmt das Betriebssystem (z. B. Windows 8 oder Linux). Wird das gestartete Programm wieder beendet, so ist der Prozess damit ebenfalls aus Betriebssystemsicht beendet. Die Ressourcen, wie z. B. vom Programm in Anspruch genommener Speicher, die ein Java-Programm während des Ablaufs belegt hat, sind durch die Beendigung des Prozesses wieder für andere Prozesse frei.

Der aufmerksame Leser wird sich nun die Frage stellen, ob sich verschiedene gestartete Prozesse auch untereinander Daten austauschen können? Selbstverständlich funktioniert das. Der dafür zuständige Speicherbereich ist der sogenannte **„Shared Memory“**.

Mehrere parallel laufende Prozesse auf einem Mikroprozessor, auch als **„Multitasking“** bezeichnet, können von den heutigen modernen Betriebssystemen wie z. B. Windows 8 gesteuert werden. Diese dafür zuständige Einheit im Betriebssystem ist der **„Scheduler“**. Er gaukelt uns lediglich nur vor, dass die Prozesse bzw. Programme auf nur einem Mikroprozessor parallel laufen. In Wirklichkeit schaltet der Scheduler alle paar Millisekunden von einem Prozess zum anderen.

Da diese Umschaltung so schnell geschieht, hat der Anwender den Eindruck, dass alle gestarteten Prozesse bzw. Programme auch auf nur einem Mikroprozessor parallel ablaufen. Diese Illusion funktioniert bei den heutigen äußerst leistungsfähigen Prozessoren natürlich absolut perfekt. Sind in einem Rechner mehrere Prozessoren eingebaut, kann das Betriebssystem alle laufenden Prozesse auf alle befindlichen Prozessoren verteilen und somit eine echte Parallelität (auch als nebenläufig bezeichnet) erreichen. Ich möchte mich in unserer Serie allerdings nur auf einen Java-Prozess auf nur einem Prozessor beschränken.

Was ist ein Thread?

Im letzten Kapitel haben wir den Begriff „Prozess“ kennengelernt. In einem Prozess wird der Programmcode sequenziell zur Ausführung gebracht. Das Betriebssystem nimmt für einen Prozess dafür mindestens einen **„Thread“** her. Thread bedeutet zu Deutsch Faden oder Ausführungsstrang. Wenn also jeder Prozess mindestens einen Thread beinhaltet, dann werden nur noch die Threads parallel ausgeführt und nicht mehr die Prozesse. Das wiederum bedeutet, dass innerhalb eines Prozesses auch mehrere Threads (auch als **„Multithreading“** bezeichnet) ablaufen können. Da ja ein Prozess ein Adressraum besitzt, müssen sich alle quasi parallel laufenden Threads denselben Adressraum teilen.

Java und Multithreading

„Multithreading“ ist eines der elementaren Merkmale von Java und wird hervorragend von der Sprache unterstützt. Java kann sogar über die JVM (Java virtual maschine) die Thread-Verwaltung auf das Betriebssystem direkt abbilden, wenn das Betriebssystem keine Threads direkt verwendet. Wir werden uns noch genauer anschauen,


```
/**
 * Datei: StartThread.java
 *
 * Dieses Programm startet einen Thread und wartet auf dessen Beendigung
 */

package com.oemus.itkolumne;

public class StartThread {

/**
 * Startet das Java-Programm
 */
public static void main (String args[]) {

  IchBinEinThread ichBinEinThread = new IchBinEinThread();

  //*******************************
  // Hier wird der Thread gestartet
  //*******************************
  ichBinEinThread.start();


  try {
    Thread.sleep(2000); // Ich warte jetzt 2 Sekunden, bevor ich den Interrupt sende!

    // Eine Unterbrechung an die Thread-Methode senden,
    // damit der Thread sich nach 2 Sekunden beenden kann!
   ichBinEinThread.interrupt();
  } catch (InterruptedException e) {
   ; // Tue nichts!
  }
 }
}

/**
 * Datei: IchBinEinThread.java
 *
 * Dieses Programm startet einen Thread und wartet auf dessen Beendigung
 */

package com.oemus.itkolumne;


public class IchBinEinThread extends Thread {

 /**
  * Diese Methode stellt die Thread-Implementierung dar
  */
 @Override
 public void run() {
  System.out.println ("Der Thread ist nun gestartet!");

  while (!isInterrupted()) {
   System.out.println("Ich laufe und laufe");

   try {
    Thread.sleep(500); // Der Thread legt sich für ½ Sekunde schlafen!
   } catch (InterruptedException e) {
    interrupt(); // Interrupt-Signal zum Beenden senden!
    System.out.println("Unterbrechung nach 2 Sekunden");
   }
  }

  System.out.println ("Der Thread ist nun wieder beendet!");
 }
}
```

Kasten 1: Java-Programm mit einem Thread.

```
Ausgabe des Programms:
Der Thread ist nun gestartet
Ich laufe und laufe
Ich laufe und laufe
Ich laufe und laufe
Ich laufe und laufe
Unterbrechung nach 2 Sekunden
Der Thread ist nun wieder beendet!
```

Kasten 2: Ausgabe des Programms.

wie mit der Programmiersprache Java in einem Programm Threads gestartet werden. Ein einmal gestarteter Thread ist prinzipiell ein Stück Java-Programmcode, der zum restlichen Programmcode parallel ausgeführt wird und somit die Geschwindigkeit des Programmablaufs massiv erhöhen kann. Was passiert aber nun, wenn z. B. der parallel zum anderen Java-Code laufende Thread konkurrierend auf eine gemeinsame Ressource (z. B. Datei) zugreift? In dieser Situation müssen entsprechende Code-Blöcke **„synchronisiert“** werden. Ein synchronisierter Code-Block verhindert den gleichzeitigen Zugriff auf eine gemeinsame Ressource im Programm. Werden solche Code-Blöcke nicht synchronisiert, kann es zu sogenannten „**Verklemmungen“** bzw. **„Deadlocks“** kommen, da ja in dieser verzwickten Situation keiner einen Vortritt bekommt, da dafür im Code nichts vorgesehen ist. Man kann hier schon gut erkennen, dass die Multithread-Programmierung in Java ein sehr kompliziertes Unterfangen sein kann und somit mit äußerster Vorsicht programmiert werden muss.

Ein Beispiel-Programm mit einem Thread

In diesem Kapitel wollen wir nun ein kleines Programm entwickeln, dass folgendermaßen funktioniert:

1. Anweisungen in der Startklasse **„StartThread“**:
   - Ein Hauptprogramm instanziiert eine Thread-Klasse mit `IchBinEinThread new IchBinEinThread();`
   - Die Anweisung `ichBinEinThread.start()`; startet die **run-**Methode der Klasse **IchBin EinThread**, die von der Klasse **Thread** abgeleitet sein muss. Jetzt läuft der Code der **run-**Methode parallel zum Hauptprogramm!
   - Mit Thread.sleep(2000) legt sich das Hauptprogramm 2 Sekunden schlafen. 2000 bedeutet 2.000 Millisekunden = 2 Sekunden. Anders ausgedrückt: Das Programm wartet 2 Sekunden lang, bevor es die nächste Anweisung ausführt.
   - Wenn die 2 Sekunden abgelaufen sind, wird das Hauptprogramm wieder aufgeweckt und sendet mit der Anweisung `ich BinEinThread. interrupt()`; ein Interrupt-Signal an die run-Methode der von Thread abgeleiteten Klasse. Damit kann sich die run-Methode beenden und das ganze Programm wird beendet. Wenn kein Interrupt-Signal an die run-Methode gesendet werden würde, könnte sich der gestartete Thread nicht beenden, was bedeutet, dass der Thread ewig lange laufen würde.
2. Anweisungen in der von Thread abgeleiteten Klasse **„IchBinEinThread“**:
   - Die Klasse muss mit von Thread abgeleitet sein, damit die **run-**Methode implementiert werden kann.
   - Die **run-**Methode stellt den Kern der Klasse dar und beinhaltet alle Anweisungen für den Teil des Programmes, der parallel zum Hauptprogramm ablaufen soll.
   - Mit der Anweiswung `while (!isInterrupted())` wird eine Schleife gestartet, die für die Beendigung ein Interrupt-Signal abfragt. Solange kein Interrupt-Signal empfangen wurde, wird der Schleifeninhalt ausgeführt.
   - Die Code-Ausführung wird dann für eine ½ Sekunde angehalten, bevor es weitergeht.
   - Sendet das Hauptprogramm ein **Interrupt-Signal** an das von Thread abgeleitete Objekt, kann die run-Methode mit einem eigenen interrupt-Signal die Schleife beenden und somit das ganze Programm nach 2 Sekunden beenden.

Im Kasten 1 sehen Sie den dazugehörigen Code der zwei benötigten Java-Dateien. Im Kasten 2 sind die vom Thread generierten Ausgaben gezeigt.

Threads mit Prioritäten

Prinzipiell hat in Java jeder Thread eine sogenannte **„Priorität“**. Diese steuert, welcher Thread wann dem Mikroprozessor für den Ablauf zugeteilt wird. Die Zuteilung wird auch als **„Scheduling“** bezeichnet. In Java gibt es insgesamt 10 Thread-Prioritäten, von 1 (niedrigste Priorität) bis 10 (höchste Priorität).

Mit der Anweisung `setPriority()` aus der Klasse **„Thread“** kann eine Priorität zugewiesen werden. Der Defaultwert (wenn keine explizite Priorität zugewiesen wird) ist `Thread.NORM_PRIORITY` (Wert = 5).

Beispiel:
Mit `setPriority(Thread.MIN_PRIORITY)`; wird dem Thread die minimalste Priorität (Wert = 1) zugewiesen.
Mit `setPriority(Thread.MAX_PRIORITY)`; wird die Priorität (Wert = 10) zugewiesen.
Mit der Anweisung `setPriority()` aus der Klasse **„Thread“** kann die Priorität abgefragt werden.

Ausblick zum Teil 10 der Serie

Im kommenden Teil zehn der Serie werden wir uns noch ausführlicher und tiefgründiger mit dem Thema **„Multithreading bzw. Nebenläufigkeit in Java“** beschäftigen. Bleiben Sie also weiter dran. ZT

ZT Autor

Thomas Burgard

Infos zum Autor

ZT Adresse

Thomas Burgard Dipl.-Ing. (FH)
Softwareentwicklung & Webdesign
Bavariastraße 18b
80336 München
Tel.: 089 540707-10
info@burgardsoft.de
www.burgardsoft.de
burgardsoft.blogspot.com
twitter.com/burgardsoft