Nebenläufigkeit

Definition, wichtige Begriffe und Vorbereitung

Nebenläufigkeit/Parallelität ist eine Eigenschaft eines Systems/Programms mehrere Berechnungen, Anweisungen oder Befehle **gleichzeitig** ausführen zu können.

Die Operationen können auf einem ***Mehrkernprozessor*** oder ***mehreren physisch getrennten Prozessoren*** oder über ein ***Netzwerk verbundener Rechner*** (-> nächstes Thema: Kommunikation zwischen Rechnern) nebenläufig ausgeführt werden.

Anwendungsbereiche:

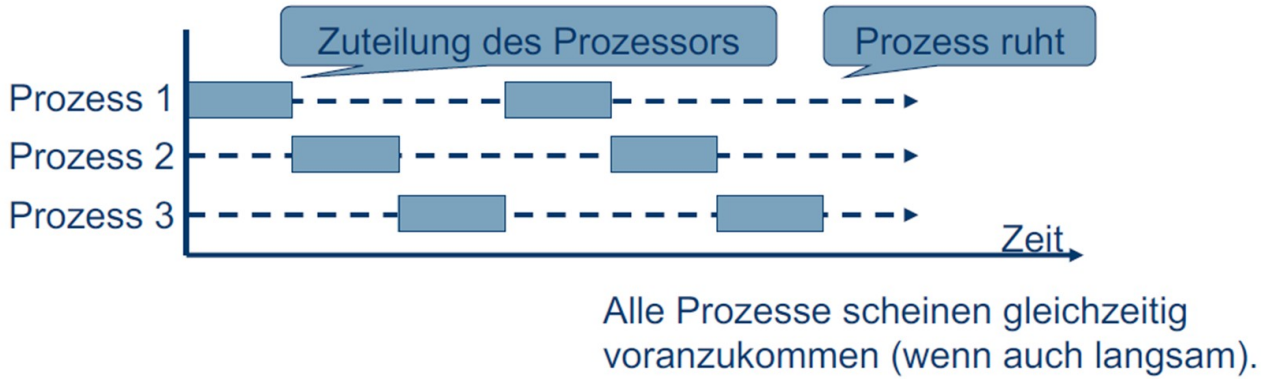
- Parallelität benötigt man immer, wenn bestimmte Ereignisse gleichzeitig ablaufen sollen. Beispiel: Musik während eines Spiels abspielen. Damit die Musik flüssig und durchgehend abgespielt werden kann und während des Abspielens das Spiel nicht pausieren soll, muss dies über Parallelisierung programmiert werden.
- Parallelität ist immer vorhanden, wenn mehrere Rechner involviert sind.
- Parallelität und Kommunikation sind die Zukunft. Beispiel: Jede Handy-App kann mittlerweile mit anderen Geräten/Internet kommunizieren und viele Apps nutzen damit auch bereits die Mehrkernprozessoren voll aus, die auch in jedem Handy vorhanden sind.

Wichtige Begriffe

- ***Prozess*** = ein im Speicher(zugriff) befindliches Programm mit seinen dafür nötigen Datenstrukturen mit Anwendungsdaten, Verwaltungsdaten (wie Methodenstapel, Programmzähler, …), Systemzustand (geöffnete Dateien, …)
- ***Ablaufplaner*** (***scheduler***) ist ein Programm/Bestandteil des Betriebssystems und teilt die Prozesse dem Prozessor bzw. Prozessoren zu, damit diese (abwechselnd) vorankommen.
- ***Thread/Aktivitätsfaden(-objekte)*** sind die logischen Einheiten/Objekte, die Code ausführen können. Jeder Prozess besitzt mindestens einen – oder auch mehrere – Threads.

Echt parallel vs. Quasi-parallel

Auf einem Rechner mit einem 1-Kern-Prozessor können auch mehrere Prozesse scheinbar gleichzeitig ausgeführt werden. Hier wird die Rechenleistung des Prozessor durch den Scheduler des Betriebssystems auf die Prozesse „verteilt“ (quasi-parallel).



Java: mehrere Klassen in einer Datei („Hilfsklassen“)

Analog zu Hilfsmethoden (z.B. bei Rekursion) ist es für die Parallelität nötig weitere Klassen zu definieren.

Diese werden aufgrund der Übersichtlichkeit im Projekt und weil sie ausschließlich für die Parallelisierung des Codes nur einer bestimmten Klasse bzw. Methode einer Klasse zuständig sind, in der gleichen Datei implementiert. Folgende Möglichkeiten existieren:

- Eine Datei kann mehrere Klassen enthalten, aber nach "außen" darf nur eine Public-Klasse sichtbar sein.
- In einer Public-Klasse dürfen allerdings weitere Public-Klassen implementiert sein.
- Anonyme Klassen werden beim Initialisieren auch definiert. Sie erweitern existierende Klassen ohne extends zu verwenden. Sie können aber nicht alleine existieren, sondern nur bestehende Klassen erweitern.

Aufgabe 1:

Sieh dir die Beispielcodes zu den Varianten der Implementierung der Hilfsklassen an.

Erzeugung von Nebenläufigkeit in Java

Es gibt viele Varianten wie Nebenläufigkeit in Java erzeugt werden kann. Im Folgenden wird allerdings nur eine Variante vorgestellt, die aber in nahezu allen Fällen von Parallelisierung eingesetzt werden kann:

- ***Runnable*** ist ein von Java vorgegebenes Interface, welches man implementieren kann (**kein Import nötig!**).
- Die Methode ***public void run()*** muss aufgrund des Interfaces implementiert werden.
- Ein Runnable ist ein ***Arbeitspaket***, welches dem Konstruktor eines Threads übergeben werden kann. ***Thread*** ist ebenfalls eine von Java vorgegebene Klasse.

- Der Code in der run-Methode des Runnable-Objekts kann parallel ausgeführt werden, indem man den Thread – der das Runnable-Objekt übergeben bekommen hat – mit **Thread.start()** aufruft. *(Bei Thread.run() wird kein neuer Aktivitätsfaden erzeugt, sondern der „main-Thread" führt diese Methode (dann sequentiell) aus.)*
- Mit **Thread.join()** kann der „main-Thread" die gestarteten, parallel laufenden Threads wieder „einsammeln" bzw. die Aktivitätsfäden wieder zusammenführen.

Aufgabe 2:

a) *Verschaffe dir einen Überblick über den Code des Beispielprogramms für Nebenläufigkeit.*

b) *Gehe den Code Schritt für Schritt durch und vollziehe die Handhabung der Parallelisierung mit Runnable-Objekten und Thread-Objekten nach.*

c) *Führe den Code mehrfach hintereinander aus. Was kannst du bezüglich der Ausgabe diesen parallelen Programms feststellen? (**sehr wichtige Erkenntnis**).*

d) *Schreibe eine neue Klasse BrokenCounter. Diese hat ein öffentliches, statisches Attribut* `public static int counter = 0;` *(Rückblick: statische Attribute und Methoden „hängen" an der Klasse und können ohne Objekt existieren. Mit Klassenname.Variable/Methode() kann auf diese zugegriffen werden). Analog zum Beispielprogramm sollen 10 Threads gestartet werden, die jeweils den counter 10 Mal um 1 erhöhen. Danach soll der Main-Thread den* `counter` *ausgeben.*

e) *Führe dein Programm mehrfach aus und erhöhe dabei die Anzahl wie oft die Threads den counter erhöhen (ca. bis 100000 pro Thread). Was fällt dir auf?*

Fehlerquellen aufgrund von Parallelität

Neben den bereits bekannten Syntaxfehler (Compiler-Fehler) und den Laufzeitfehlern kommen bei parallelen Programmen weitere Fehlerquellen hinzu, die aufgrund ungünstiger Verzahnung von Codeabschnitten verschiedener Threads entstehen können.

Fehler: Wettlaufsituation (Race condition)

→ Siehe Präsentation Wettlaufsituation

Das Zähler-Beispiel zeigt die klassische „***Lesen-Ändern-Schreiben-Wettlaufsituation***". Sie kann immer dann auftreten (= notwendige Bedingung),

- wenn eine Variable von mehr als einer Aktivität gelesen, abhängig vom Wert geändert und dann das Ergebnis zurück in die Variable geschrieben wird, also
- wenn Lesen, Ändern und Schreiben nicht als **ununterbrechbare Einheit (atomare Aktion)** geschehen.

Schlüsselwort synchronized

Jedes Objekt in Java besitzt genau eine **Marke (Token)**, die sich ein Thread mittels ***synchronized*** exklusiv für sich beanspruchen kann. Nur der eine Thread der diese Marke besitzt darf den ***kritischen Abschnitt*** „betreten“ und den Code ausführen. Die Threads, die die Marke nicht bekommen haben werden am synchronized-Block ***blockiert*** und warten bis sie die Marke bekommen. Am Ende des Codes wird dann diese Marke wieder freigegeben, sodass der nächste Thread den kritischen Abschnitt ausführen kann. Somit wird der kritische Abschnitt sequentiell ausgeführt.

```
//hier warten die Threads auf die Marke
synchronized(Referenz_auf_ein_Objekt){
    //kritischer Abschnitt
}
```

Den synchronized-Block möchte man möglichst „klein“ halten, da sonst der „Speed Up“ durch die Parallelisierung verloren geht.

Falls nur eine einzige Variable synchronisiert werden muss, bietet Java die Klassen aus ***java.util.concurrent.atomic*** an. Die angebotenen Methoden der atomic-Klassen (get(), set(…), getAndIncrement(), getAndAdd(…), compareAndSet(…)) sind werden **atomar** ausgeführt und sind somit **thread-sicher**!

Vorteil: Man kommt ohne aufwändiges Blockieren von Aktivitätsfäden aus, da die Methoden auf ununterbrechbare Hardware-Instruktionen abgebildet werden können.

Aufgabe 4:

a) *Picasso*

b) *Monte-Carlo-Integration (Präsentation im Code-Ordner) (für schnelle Schüler)*

Fehler: Verklemmung (Deadlock)

→ siehe Präsentation Verklemmung

Eine ***Verklemmung*** ist ein schwerwiegender Fehler in einem Programm, bei dem ein Aktivitätsträger nicht weiterarbeiten, weil er auf irgendetwas wartet, das nicht eintreten

wird bzw. nicht verfügbar wird. → Daher muss der Programmierer sicherstellen, dass Verklemmungen gar nicht erst eintreten.

Folgende Voraussetzungen sind notwendig, dass eine Verklemmung eintritt:

- ***Gegenseitiger Ausschluss***
- ***Kein Entzug***
- ***Iterative Anforderung***

Die zusätzliche Bedingung, sodass eine Verklemmung auftritt ist die ***zirkuläre Abhängigkeit*** (engl. deadly embrace).

Aufgabe 5:

a) *Philosophen-Problem (Präsentation Verklemmung)*

b) *Erzeuger-Verbraucher-Problem (gemeinsam)*