17. Parallele Programmierung in Java mit Threads

- Überblick
  - Klassische Verwendung von Threads
  - Synchronisation
  - ExecutorService, Callable und Future

Motivation

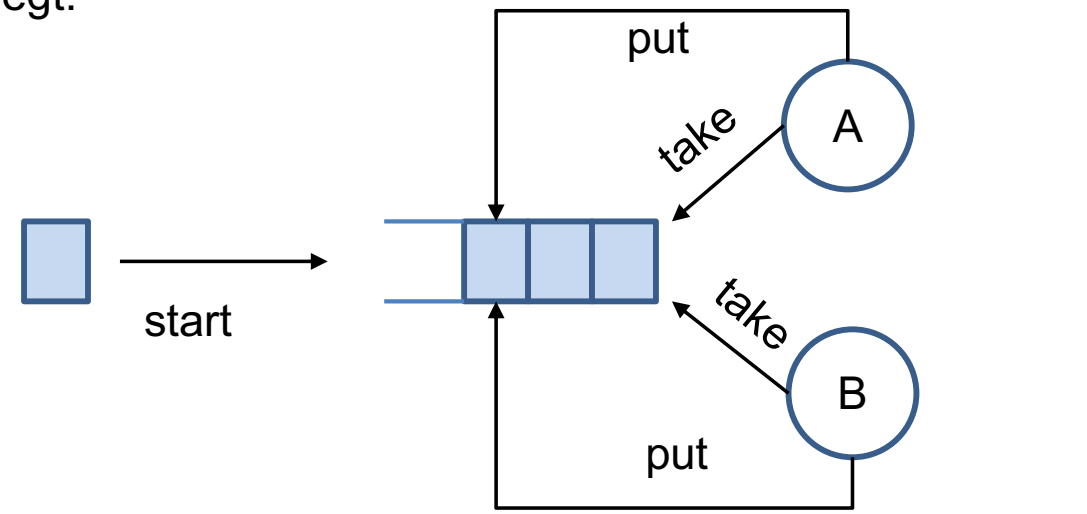
- Komplexe Programmsysteme müssen gleichzeitig verschiede Benutzer bedienen (Pseudo-Parallelität).
  - Verschiedene Operationen laufen gleichzeitig miteinander (auch wenn nicht notwendigerweise mehrere CPUs vorhanden sind).
  - Beispiele
    - Betriebssysteme
    - Datenbanksysteme
- Parallele Rechnersysteme
  - Eine Aufgabe soll parallel programmiert werden, um alle Kerne eines Rechners zu nutzen und somit die Laufzeit eines Programms zu verringern.
    - Parallelisieren von klassischen Algorithmen wie Sortieren, Suchen, …

Ausführungsthreads

- Eine moderne CPU besitzt mehrere Kerne
  - Durch Simultaneous Multi-Threading (SMT) kann jeder Kern mehrere Instruktionen parallel ausführen
  - Typische Desktop CPUs haben 4-8 Kerne, die jeweils 2 SMTs ausführen können
  - D.h., 8-16 Hardware-Threads
- Oft benötigen Anwendungen mehr Ausführungsthreads
  - Außerdem werden mehrere Anwendungen gleichzeitig ausgeführt
- Das Betriebssystem und die Programmiersprache unterstützen durch Software-Threads wesentlich mehr Ausführungsthreads
  - Größenordnung mehrere Tausend bis Zigtausend

Realisierung von Software-Threads durch Betriebssystem/ Java Virtuelle Maschine

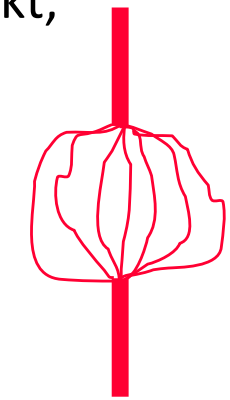
- Threads werden nach dem Start in einer Queue abgelegt.
- Prozessoren nehmen sich die Threads aus der Queue und verarbeiten diese bis ein Ereignis, wie z. B. das Zeitlimit ist überschritten, eintritt.
  - Falls Thread noch nicht fertig ist, wird dieser unterbrochen und in die Queue abgelegt.



17.1 Threads

- Bisher
  - Ein Java-Programm hat zu jedem Zeitpunkt genau eine aktive Methode.

- Threads erlauben die gleichzeitige Verarbeitung von mehreren aktiven Methoden in einem Programm.
  - Der Programmablauf verläuft gemeinsam bis zu einem gewissen Punkt, dann teilt sich die Ausführung in mehrere Threads.
  - Zu einem späteren Zeitpunkt kann die Ausführung dann wieder gemeinsam fortgesetzt werden.



- Wichtige Klassen und Interfaces in Java
  - funktionales Interface Runnable
  - Klasse Thread

Interface Runnable

- Funktionale Schnittstelle bietet die Methode void run().
  - In der Methode wird die gewünschte Funktionalität implementiert, die über ein Objekt der Klasse Thread parallel ablaufen soll.
- Zwei Möglichkeiten
  - Implementierung über eine eigene Klasse

```
class TicTacToe implements Runnable{
    String was;
    TicTacToe(String s){ was = s; }
    public void run(){
        System.out.println(was);
    }
}
```

  - Üblicher: Implementierung als anonyme Klasse oder Lambda

```
Runnable tic = () -> System.out.println("Tic");
```

Klasse Thread

- Die Klasse Thread wird benutzt, um ein Runnable-Objekt parallel auszuführen.
  - Erzeugung eines Objekts der Klasse mit einem Runnable-Objekt

```
Thread t = new Thread(tic);
```

  - Starten des Threads mit der Methode start()

```
t.start();
```

  - Danach gibt es zwei aktive Methoden (Ausführungsfäden, Threads)
    - Hauptprogramm wird fortgeführt.
      - Dieser Ausführungsfaden endet mit der Methode main.
    - Methode start führt die Methode run des Objekts tic aus.
      - Dieser Ausführungsfaden endet mit der Methode run.
    - Beide laufen voneinander unabhängig.

Beispiel

- Betrachten wir folgendes Programm:

```
public static void main(String args[]) {
    System.out.println("Thread Beispiel");
    Runnable tic = () -> System.out.println("Tic ");
    Runnable tac = () -> System.out.println("Tac ");;
    Runnable toe = () -> System.out.println("Toe ");;
    new Thread(tic).start();
    new Thread(tac).start();
    new Thread(toe).start();
    System.out.println("Hauptprogramm");
}
```

1. Programmstart z.B.:

```
Thread Beispiel
Hauptprogramm
Tac
Tic
Toe
```

- Die Ausgabe ist nichtdeterministisch.
  - Das gleiche Programm kann ganz verschiedene Ausgaben produzieren

2. Programmstart z.B.:

```
Thread Beispiel
Tic
Toe
Tac
Hauptprogramm
```

Weitere Methoden der Klasse Thread

- static void sleep(long millis)
  - Beim Aufruf dieser Methode wird die aufrufende Methode für *millis* Millisekunden schlafen gelegt.
- void setPriority(int newPriority)
  - Hiermit kann die Priorität eines Threads auf newPriority gesetzt werden.
    - Ein Thread mit hoher Priorität kommt öfters auf einem Prozessor zur Ausführung als ein Thread mit niedriger Priorität.
- void join()
  - An dieser Stelle wird gewartet bis der zuvor gestartete Thread endet.

Beispiel

```
public static void main(String args[]) {
    System.out.println("Thread Beispiel");
    Runnable tic = () -> System.out.println("Tic ");
    Runnable tac = () -> System.out.println("Tac ");
    Runnable toe = () -> System.out.println("Toe ");
    Thread[] t = {new Thread(tic), new Thread(tac), new Thread(toe)};
    for (i=0; i < 2; i+=1)
        t[i].start();
    for (i=0; i < 2; i+=1)
        try {
                t[i].join();
        }
        catch (InterruptedException ie ){ /* toDo */ }
    System.out.println("Hauptprogramm");
}
```

```
Thread Beispiel
Tic
Toe
Tac
Hauptprogramm
```

- Die Ausgabe ist immer noch nichtdeterministisch.
  - Jedoch ist garantiert, dass stets am Schluss **"Hauptprogramm"** ausgegeben wird.

```
Thread Beispiel
Hauptprogramm
Tac
Tic
Toe
```

17.2 Synchronisation

```
public class SyncProblems {
    static int count = 0;
    static void increment() {
        int countTemp = count;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        count = countTemp + 1;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[100];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(SyncProblems::increment);
            threads[i].start();
        }
        for (Thread thr: threads) {
            thr.join();
        }
        System.out.println(count);      // Was ist das Ergebnis?
    }
}
```

17.2 Synchronisation

```
public class SyncProblems {
    static int count = 0;
    static void increment() {
        int countTemp = count;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        count = countTemp + 1;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[100];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(SyncProblems::increment);
            threads[i].start();
        }
        for (Thread thr: threads) {
            thr.join();
        }
        System.out.println(count);      // Was ist das Ergebnis?
    }
}
```

Lost Update Problem

Ändern Threads in parallel eine Variable, können einige Änderungen verloren gehen.

Schlüsselwort synchronized

- Dieses Problem kann durch Verwendung von synchronized behoben werden.

```
static synchronized void increment() {
    int countTemp = count;
    try {
            Thread.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    count = countTemp + 1;
}
```

- Damit wird sichergestellt, dass nur ein Thread zu einem Zeitpunkt die Methode ausführen kann.
  - Man kann auch einzelne Code-Blöcke mit dem Schlüsselwort synchronized schützen.
- Der Code wird dann auch als Thread-safe bezeichnet.

17.3 Parallele Streams (java.util.stream)

- Eine Pipeline beschreibt
  - was berechnet,
  - aber nicht wie etwas berechnet werden soll.
- Die Iteration der Elemente wird intern implementiert
  - for-Schleife vs. forEach
  - Optimierungen möglich unter Ausnutzung interner Strukturen
  - einfache Parallelisierung

```
double average = Stream.of(5, 4, 7, 2, 10, 8).parallel()
    .filter(x -> {
        System.out.println("Filter: " + x);
        return x % 2 == 0;
    }).collect(Collectors.averagingInt(v -> v));
System.out.println("Durchschnitt: " + average);
```

```
Filter: 8
Filter: 7
Filter: 5
Filter: 10
Filter: 4
Filter: 2
Durchschnitt:
6.0
```

```
Filter: 8
Filter: 10
Filter: 7
Filter: 4
Filter: 2
Filter: 5
Durchschnitt:
6.0
```

17.4 Executor-Schnittstelle

- Problem bei der Thread-Programmierung ist oft die Strategie und Implementierung für die Ablaufsteuerung.
  - Da die Erzeugung der Thread-Objekte teuer ist, sollten Threads wiederverwendet werden
- Vorgefertigte Strategien werden als Objekte des Interface ExecutorService im Package java.util.concurrent zur Verfügung gestellt.
  - Die Objekte werden über statische Methoden aus der Klasse Executors geliefert.

Thread-Erzeugung mittels ExecutorService

- Übergabe der run-Methode an submit-Methode von ExecutorService
- Ein Executor wird zunächst gestartet.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
executorService.submit(() -> System.out.println("Tic "));
executorService.submit(() -> System.out.println("Tac "));
executorService.submit(() -> System.out.println("Toe "));
executorService.shutdown();
```

Lambdas entsprechen der funktionalen Schnittstelle Runnable.

- Der Executor muss am Ende des Programms noch runtergefahren werden.
  - Aufruf der Methode executor.shutdown()

Schnittstelle Callable

- Java bietet inzwischen die Schnittstelle Callable an, um Threads zu erzeugen, die ein Ergebnis zurückliefern.
  - Callable-Objekte können ebenfalls über die Methode submit an ein ExecutorService-Objekt übergeben werden.

```
Callable<Integer> task = () -> {
        try {
            Thread.sleep(10);
            return 42;
        }
        catch (InterruptedException e) {
            throw new IllegalStateException("task interrupted", e);
        }
    };
ExecutorService executorService = Executors.newSingleThreadExecutor();
```

Callable ist ebenfalls eine funktionale Schnittstelle mit einer ähnlichen Bedeutung wie Runnable. Im Unterschied hierzu hat die Funktion aber einen Rückgabetyp.

- Wie kommt man an das Ergebnis des Threads ran?

Schnittstelle Future

- Hierfür steht die Schnittstelle Future zur Verfügung.
  - Beim submit-Aufruf mit einem Callable wird ein Future-Objekt geliefert.
- Das Future-Objekt gibt uns mit der Methode get das gewünschte Ergebnis.
  - Falls das Ergebnis noch nicht produziert wurde, wird an dieser Stelle gewartet bis das Ergebnis vorliegt.

```
...
//
Future<Integer> future = executorService.submit(task);
System.out.println("future done? " + future.isDone());
Integer erg = 0;
try {
   erg = future.get();
}
catch (Exception e) { /* toDo */ }
System.out.println("Future fertig? " + future.isDone());
System.out.print("Resultat: " + erg);
```

Zusammenfassung

- Nebenläufige und parallele Verarbeitung durch Threads
- Erzeugen und Ablaufen von Threads
  - Executor
  - Schnittstelle Runnable und Callable
- Synchronisation von Threads
  - Nutzen gemeinsamer Speicher
- Viele Konzepte aus Java für Threads und Synchronisation wurden in dieser kurzen Übersicht nicht behandelt.
  - ➔ Systemsoftware und Rechnerkommunikation