Programmierparadigmen

Kapitel 4d
Parallele Programmierung in Java

(Diese Folien beruhen auf einem Foliensatz von Prof. Dr. Kai Uwe Sattler)

Überblick

- Grundlagen
- Parallele Programmierung in Erlang
- Parallele Programmierung in C++
- Parallele Programmierung in Java
  - Threads
  - Wechselseitiger Ausschluss
  - Tasks, Futures und Executor

Parallele Programmierung in Java

- Unterstützung durch:
  - Thread-Konzept
  - Mechanismen zur Synchronisation nebenläufiger Prozesse
  - Spezielle High-Level-Klassen im Package `java.util.concurrent`

Threads in Java

- Repräsentiert durch Klasse `java.lang.Thread`
- Implementierung eines eigenen Kontrollflusses
  - Implementierung des Interface `java.lang.Runnable`
    - Keine weitere Beeinflussung des Threads über zusätzliche Methoden notwendig
    - Einsetzbar, wenn von anderer Klasse als Thread abgeleitet werden soll
  - Subklasse von `java.lang.Thread`
    - Zusätzliche Methoden zur Steuerung des Ablaufs benötigt
    - Keine andere Superklasse notwendig

Threads: Runnable-Schnittstelle

- Eigene Klasse muss Schnittstelle `Runnable` implementieren
  - Implementiert Methode `public void run()` die beim Start des Threads aufgerufen wird

```
1  public class Heartbeat implements Runnable {
2    int pulse;
3    public Heartbeat(int p) { pulse = p * 1000; }
4    public void run() {
5      while(true) {
6        try { Thread.sleep(pulse); }
7        catch(InterruptedException e) {}
8        System.out.println("poch");
9        }
10     }
11 }
```

Thread-Erzeugung

- Thread Objekt mit `Runnable`-Objekt erzeugen
  - Methode `start()` aufrufen, die Methode `run()` aufruft

```
1  public static void main(String[] args) {
2    Thread t = new Thread(new Heartbeat(2));
3    t.start();
4  }
```

Threads: Subklasse von Thread

- Klasse muss von Klasse Thread abgeleitet werden
- Methode `run()` muss überschrieben werden

```
1  public class Heartbeat2 extends Thread {
2    int pulse = 1000;
3    public Heartbeat2() {}
4    public void setPulse(int p) { pulse = p * 1000; }
5
6    public void run() {
7      while(true) {
8         try { Thread.sleep(pulse); }
9         catch(InterruptedException e) {}
10        System.out.println("poch");
11      }
12   }
13 }
```


Thread-Erzeugung

- Objekt der eigenen Thread-Klasse erzeugen
  - Methode `start()` aufrufen, die Methode `run()` aufruft

```
1  public static void main(String[] args) {
2    Heartbeat2 t = new Heartbeat2(2);
3    t.start();
4  }
```

- Spätere Beeinflussung durch andere Threads möglich

```
1    ...
2    t.setPulse(3);
```


Threads: Wichtige Methoden

- `void start()`
  - Initiiert Ausführung des Threads durch Aufruf der Methode `run()`
- `void run()`
  - Die eigentliche Arbeitsmethode
- `static void sleep(int millis)`
  - Hält die Ausführung des aktuellen Threads für `millis` Millisekunden an
  - Kein Einfluss auf andere Threads!
- `void join()`
  - Blockiert den aufrufenden Thread so lange, bis der aufgerufene Thread beendet ist

Parallele Berechnung von *n* Fibonacci-Zahlen (1)

- Hier über Implementierung des Runnable-Interface:

```
1  public class Fibonacci implements Runnable {
2    int fi;
3    public Fibonacci(int f) { fi = f; }
4    int fibo(int f) { // serielle Berechnung einer Zahl
5      if (f < 2) return 1;
6      else return fibo(f-1) + fibo(f-2);
7    }
8
9    public void run() {
10     int res = fibo(fi);
11     System.out.println("Fibonacci(" + fi
12      + ") = " + res);
13   }
14 }
```


Parallele Berechnung von *n* Fibonacci-Zahlen (2)

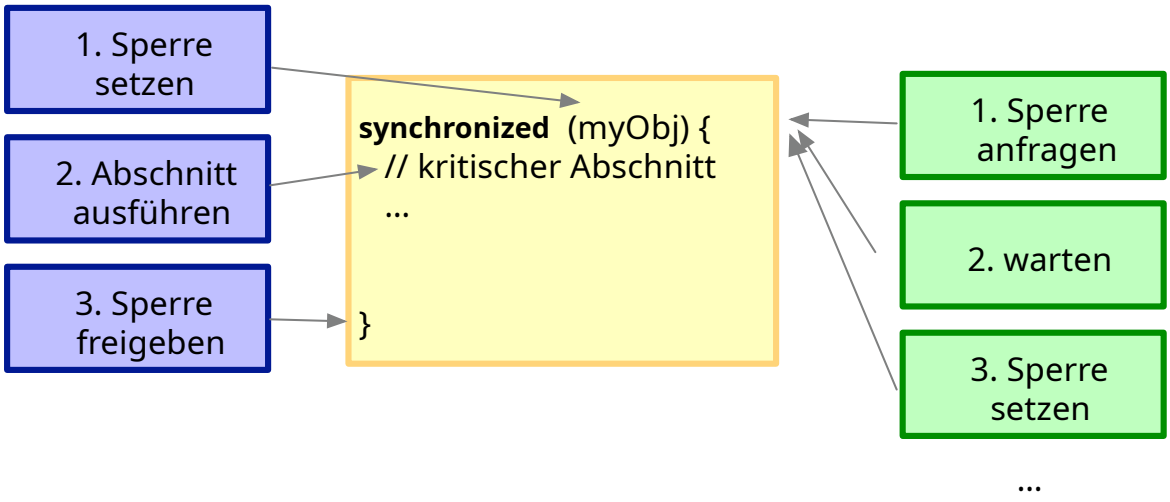
- Thread-Erzeugung und Ausführung:

```
1  public static void main(String[] args) {
2    Thread[] threads = new Thread[10];
3    for(int i = 0; i < 10; i++) { // hier n = 10
4      threads[i] = new Thread(new Fibonacci(40 + i));
5      threads[i].start();
6    }
7  }
```


Wechselseitiger Ausschluss in Java (1)

- Schlüsselwort `synchronized`
  - Implementierung von sogenannten Monitoren bzw. Locks (exklusiven Sperren)
    - Nur ein Thread darf den kritischen Abschnitt betreten
    - Alle anderen Threads, die darauf zugreifen wollen, müssen auf Freigabe warten
  - Für Methoden: `public synchronized void doSomething()`
    - Nur ein Thread darf diese Methode auf einem Objekt zur gleichen Zeit ausführen
  - Für Anweisungen: `synchronized(anObject) { ... }`
    - Nur ein Thread darf den Block betreten
    - Sperre wird durch das Objekt `anObject` verwaltet (jedem Java-Objekt ist ein Sperre zugeordnet)

Wechselseitiger Ausschluss in Java (2)



Wait & Notify

- Signalisierung zwischen Threads in Java
- Basismethoden der Klasse `java.lang.Object`
- `wait()`: der aktive Thread wartet an diesem Objekt, Sperren werden ggf. freigegeben!
- `notify()`: weckt an diesem Objekt wartenden Thread auf
- `notifyAll()`: weckt alle an diesem Objekt wartenden Threads auf
- `wait()` & `notify()` dürfen nur in einem synchronized-Block aufgerufen werden

Java High-Level-Klassen

- Paket `java.util.concurrent` seit Java Version 1.5
- Abstraktionsschicht versteckt Details über Thread-Erzeugung
- Übernimmt Erstellung und Überwachung von parallelen Tasks, u.a.
  - `ExecutorService` zum Erzeugen asynchroner Tasks
  - `Future`: Referenz auf diesen Task bzw. dessen Ergebnis
  - `ForkJoinPool` & `RecursiveAction`: rekursives Aufteilen eines großen Problems

Tasks und Threads in Java

- `Task` = Logische Ausführungseinheit
- `Thread` = Mechanismus zur asynchronen/parallelen Ausführung von Tasks

```
1  Runnable task = () -> {  // Java way of declaring Lambdas
2    String me = Thread.currentThread().getName();
3    System.out.println("Hallo " + me);
4  };
5
6  task.run(); // not parallel: runs in same thread
7
8  Thread thread = new Thread(task);
9  thread.start(); // calls task.run() in new thread
```


Runnables, Callables & Futures in Java

- Callables repräsentieren wie Runnables logische Ausführungseinheiten (= Tasks), die durch mehrere Threads parallel ausgeführt werden können
- Die Methode `run()` eines Runnable kann keine Parameter entgegennehmen und liefert keinen Ergebniswert
- Die Methode `call()` eines Callable nimmt ebenfalls keine Parameter entgegen, kann jedoch einen Ergebniswert (unterstützt Generics für Ergebnistyp) liefern und zusätzlich auch Exceptions hochreichen
- Futures erlauben es, Ergebnisse von einem gestarteten Task entgegen zu nehmen, das erst in der Zukunft vorliegen wird und dann abgefragt werden kann:
  - Abfrage mittels `get()` führt ggf. zu Blockieren/Warten bis das Ergebnis vorliegt
  - Ein Future, das mit einem Runnable erzeugt wird, liefert als Ergebnis immer `void`

Future & ExecutorService

- `ExecutorService` stellt Methoden zum Starten, Beenden und Steuern von parallelen Aufgaben bereit
- Implementiert `Executor`-Interface
  - Definiert Methode `void execute(Runnable r)`
- Starten einerAufgabe mit `submit`
  - `Future<T> submit(Callable c)`
  - `Future<?> submit(Runnable r)`
- Zugriff auf das Ergebnis mit Funktion `get`
  - `T get(long timeout, TimeUnit unit)`
  - `T get()`
- Testen ob fertig mit Funktion `isDone`
  - `boolean isDone()`

Future & ExecutorService: Beispiel

```
1   class App {
2       ExecutorService executor = Executors.newFixedThreadPool(4);
3       void doIt(final String w) throws InterruptedException {
4       Future<String> future =
5           executor.submit(new Callable<String>() {
6               public String call() {
7                   String result = w + ...; // lengthy computation
8                   return result; }
9           }); // of executor.submit() call
10      displayOtherThings(); // do other things
11      while(!future.isDone()) { ...; Thread.sleep(200); } // more
12          try {
13            displayText(future.get()); // get is blocking
14          } catch (ExecutionException ex) {
15              cleanup();
16              return; } // of catch
17      } // of doIt()
18  }
```


RecursiveAction & Fork/Join

- Rekursives Zerlegen eines großen Problems in kleinere Probleme
- Solange bis Probleme klein genug sind, um effizient seriell berechnet werden zu können
- Jeder Task erstellt rekursiv zwei oder mehr Teiltasks von sich selbst

  ⇝ **Datenparallelität**

- `ForkJoinPool` zum Ausführen
  - Implementiert `Executor` Interface
  - `<T> T invoke(ForkJoinTask<T> task)` berechnet Task bis das Ergebnis vorliegt
  - `<T> List<Future<T>> invokeAll(…)` startet mehrere Tasks, Ergebnisse können über Liste von Futures ermittelt werden

Fork/Join: Beispiel (1)

```
1  class MyTask extends RecursiveAction {
2    String[] source; // We will operate on an array of Strings
3    int start, length;
4    public MyTask(String[] src, int s, int l) {
5      source = src; start = s; length = l;
6    } // of MyTask()
7
8    void computeDirectly() { ... } // Work on a couple of Strings
9
10   @Override
11   void compute() {
12     if (length < THRESHOLD) computeDirectly();
13     else { // Here we don't need the list of Result-Futures
14           int split = length / 2;
15           invokeAll(new MyTask(source, start, split),
16             new MyTask(source, start + split, length – split));
17         }
18   } // of compute()
19 }
```


Fork/Join: Beispiel (2)

- Starten der Verarbeitung:
  1. Große Gesamtaufgabe erstellen
  2. ForkJoinPool erstellen
  3. Aufgabe vom Pool ausführen lassen

```
1  String[] src = ... // Großes Array (“Aufgabe”) erzeugen
2
3  MyTask t = new MyTask(src, 0, src.length);
4  ForkJoinPool pool = new ForkJoinPool();
5  pool.invoke(t); // calls compute() on t, waits until done
6  // From now on, resulting array src can be evaluated
7  ...
```


Fazit

- Parallelprogrammierung in Java ähnlich zu C++
- Konzepte: Threads, kritische Abschnitte über `synchronized`
- Mächtige Abstraktionen in `java.util.concurrent`
  - `Tasks` und `Futures`, `Executor` und `ThreadPool`
  - Thread-sichere Datenstrukturen
  - Synchronisation: Barrieren, Semaphoren, ...

Zusammenfassung zu Kapitel 4

- Parallelprogrammierung als wichtige Technik zur Nutzung moderner Hardware (Multicore, GPU, ...)
- Verschiedene Architekturen und Programmiermodelle
- Instruktions-, Daten- und Taskparallelität
- Message Passing vs. gemeinsamer Speicher
- Konzepte in Erlang, C++ und Java
- Hoher Abstraktionsgrad funktionaler Sprachen
- C++/Java: Thread-Modell und Synchronisation mit vielen weiteren Konzepten
- Höherwertige Abstraktionen durch zusätzliche Bibliotheken und Programmierschnittstellen