PK

Nebenläufige Programmierung

Parallelität und Nebenläufigkeit

Parallelität: gleichzeitige Ausführung von Programm(teil)en

Ziel: Leistungssteigerung, Ausnutzung der Hardware;
Varianten: feingranulär bis grobkörnig

Parallelisierung: Aufspaltung des Programms in parallel ausführbare Einheiten

Ziel: Leistungssteigerung, Ausnutzung der Hardware;
Varianten: automatisch oder manuell

Nebenläufigkeit: Programmteile scheinen gleichzeitig ausgeführt zu werden

Ziel: rasche Reaktion auf Ereignisse (durch Programmstrukturierung);
Varianten: Parallelität möglich, aber nicht notwendig

Techniken bei Nebenläufigkeit

Multiprocessing: mehrere Prozesse gleichzeitig

Prozess = Ausführung eines Programms;
jeder Prozess hat eigenen Namensraum;
Prozess aufspannen = Programm aufrufen

Multithreading: mehrere Threads pro Prozess

Thread = Ausführungsstrang innerhalb eines Prozesses;
Threads greifen auf gemeinsame Variablen und Objekte zu;
gegenseitige Behinderungen möglich

Race-Conditions

Ergebnisse hängen von Geschwindigkeit einzelner Threads ab

Beispiel MultithreadingTest:

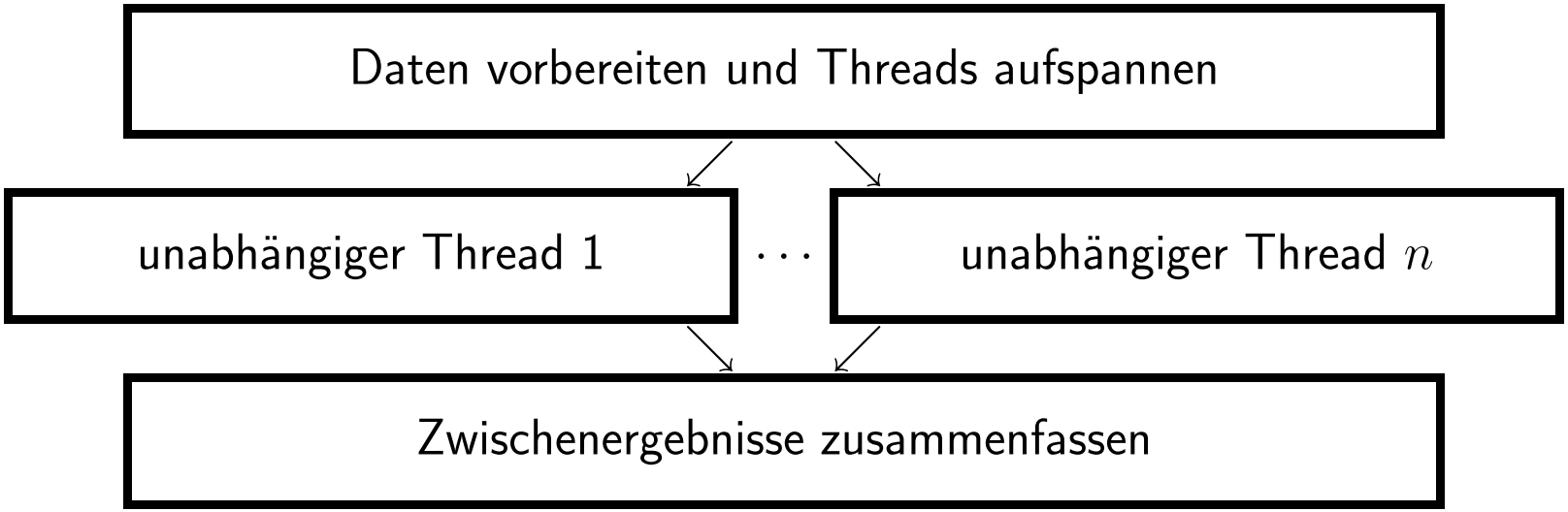
mehrere Threads führen `x++` bzw. `y++` fast gleichzeitig aus
(z.B.: lese 3 – lese 3 – schreibe 4 – schreibe 4)

Thread zwischen Lesen und Zurückschreiben unterbrochen

`x != y` wenn `x` schon erhöht, `y` noch nicht

Ausgabe dauert länger, mehrere Threads erkennen `x != y`

Threads auf getrennten Daten



Beispiel MultithreadingTest: jeder Worker hat eigenen Counter

Aufgabe: Unabhängigkeit

Such Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Fragen:

Warum sollen die Threads voneinander unabhängig sein?
Was bedeutet „voneinander unabhängig sein“?

Zeit: 2 Minuten

Atomare Aktion

```
public synchronized void increment() { ... }
```

vermeidet Race Conditions

sequentielle Ausführungen von `increment`

rasche Termination, sonst keine Parallelität

Komplexere Synchronisation

`a.wait()` versetzt aktuellen Thread in Wartezustand
(in der Warteschlange von Objekt `a`, häufig `this`)

Thread kann jederzeit auch ohne Grund aufgeweckt werden
(daher bei Bedarf `a.wait()` wiederholt ausführen)

Ausnahme `InterruptedException` muss abgefangen werden

`a.notify()` sollte beliebigen in `a` wartenden Thread aufwecken

`a.notifyAll()` weckt alle in `a` wartende Threads auf

Gegenseitige Behinderung

bei Programmkonstruktion peinlich auf Vermeidung von Race-Conditions achten!

Probleme vergleichbar mit Straßenverkehrsregeln

Gefährliche Situationen trotz atomarer Aktionen:

- Starvation
- Livelock
- Deadlock

solche Situationen durch ausgiebiges Testen erkennbar