
Nebenläufige Programmierung

Parallelität und Nebenläufigkeit

Parallelität: gleichzeitige Ausführung von Programm(teil)en

- Ziel: Leistungssteigerung, Ausnutzung der Hardware
- feingranulär bis grobkörnig

Parallelisierung: Aufspaltung eines Programms in weitgehend unabhängige, parallel ausführbare Einheiten

- Ziel: Leistungssteigerung, Ausnutzung der Hardware
- automatisch oder manuell

Nebenläufigkeit: Strukturierung des Programms so als ob Programmteile gleichzeitig ausgeführt würden

- Ziel: rasche Reaktion auf Ereignisse (Programmierstil)
- 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
- greifen auf gemeinsame Variablen und Objekte zu
- gegenseitige Behinderungen möglich

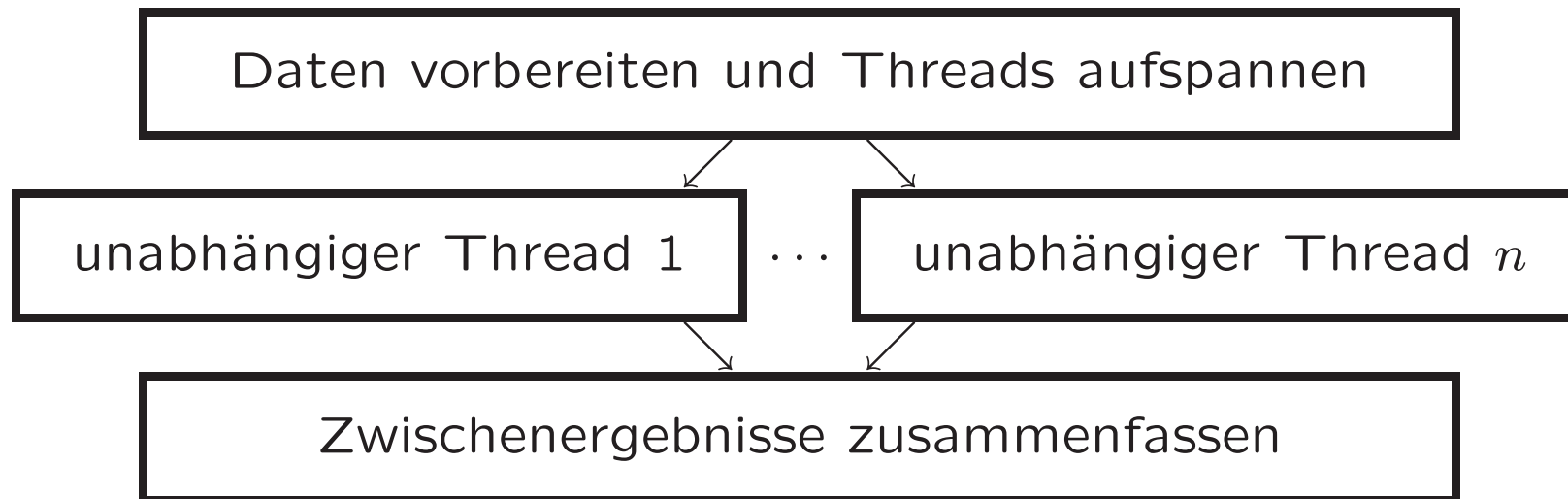
Race Conditions

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
- $x \neq y$ wenn x schon erhöht, y noch nicht
- Ausgabe dauert länger, mehrere Threads erkennen $x \neq y$

Threads auf getrennten Daten



Beispiel MultithreadingTest: jeder Worker hat eigenen Counter

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()` weckt beliebigen in `a` wartenden Thread auf
- `a.notifyAll()` weckt alle in `a` wartenden Threads auf

Gegenseitige Behinderung

- bei Programmkonstruktion peinlich auf atomare Aktionen zur Vermeidung von Race-Conditions achten!
- Probleme vergleichbar mit Straßenverkehrsregeln
- Gefährliche Situationen trotz atomarer Aktionen:
 - Starvation
 - Livelock
 - Deadlock
- solche Situationen durch ausgiebiges Testen erkennbar