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

Schizophrener Zähler

```
class Counter {
    private int x = 0, y = 0;

    public void increment() {
        if (x != y) {
            System.out.println("x = " + x + "; y = " + y);
            x = y = 0;
        }
        x++;
        y++;
    }
}
```

Aufspannen von Threads

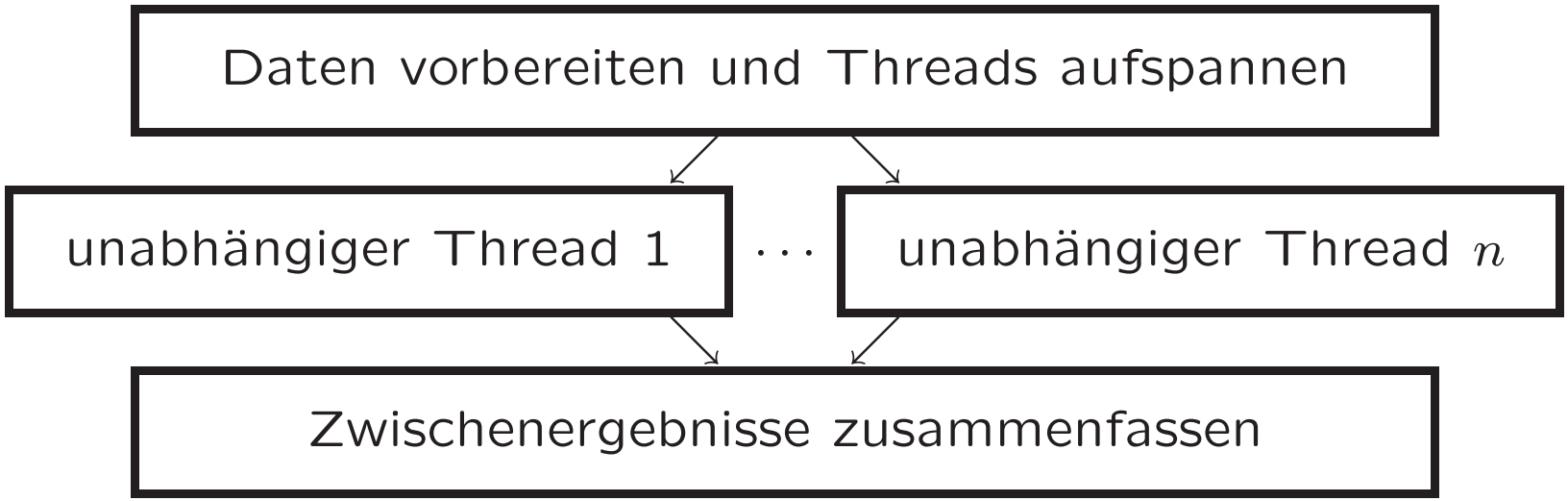
```
class Worker implements Runnable {
    private Counter counter;
    public Worker (Counter c) { counter = c; }
    public void run() {
        for(int i = 0; i < 100000; i++)
            counter.increment();
    }
}
public class TestMultithreading {
    public static final void main (String[] args) {
        Counter c = new Counter();
        for(int i = 0; i < 10; i++)
            new Thread(new Worker(c)).start();
    }
}
```

Race Conditions (Beispiel)

Ergebnisse hängen von Geschwindigkeit einzelner Threads ab:

- 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



Atomare Aktion

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

- vermeidet Race Conditions
- sequentielle Ausführungen von `increment`
- rasche Termination, sonst keine Parallelität

Synchronisierter Zähler

```
class Counter {
    public static final int BARRIER = 200000;
    private int x = 0, y = 0;
    public synchronized void increment() {
        x++; y++;
        if (x == BARRIER) notifyAll();
    }
    public synchronized void wow() {
        while (x < BARRIER) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        System.out.println("Wow! Schon bei " + x + "!");
    }
}
```

Test für Synchronisierten Zähler

```
public class TestWaiting {
    public static final void main (String[] args) {
        Counter counter = new Counter();
        for(int i = 0; i < 10; i++)
            new Thread(new Worker(counter)).start();
        for(int i = 0; i < 3; i++)
            counter.wow();
    }
}
```

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