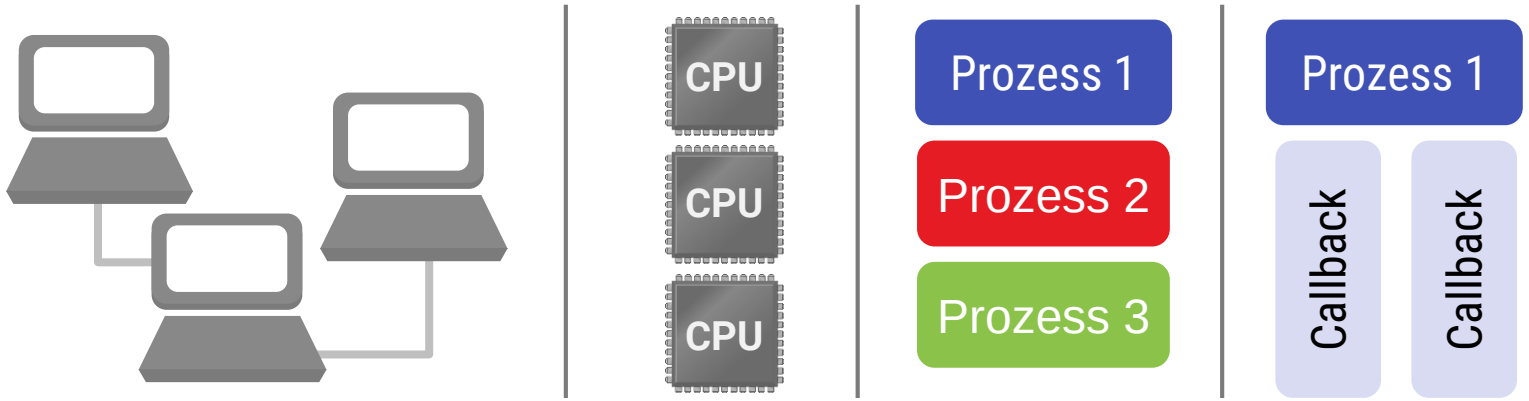
# Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020




Foliensatz #13

# Nebenläufigkeit & Design

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

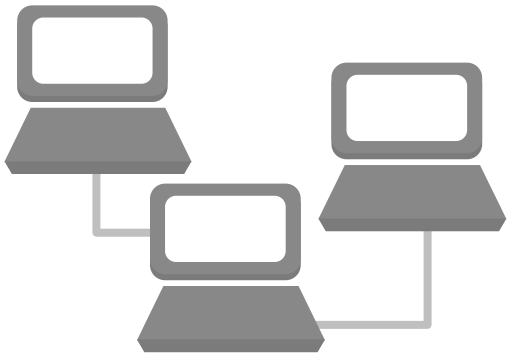
# Parallele Programmierung

# Übersicht

## Themen

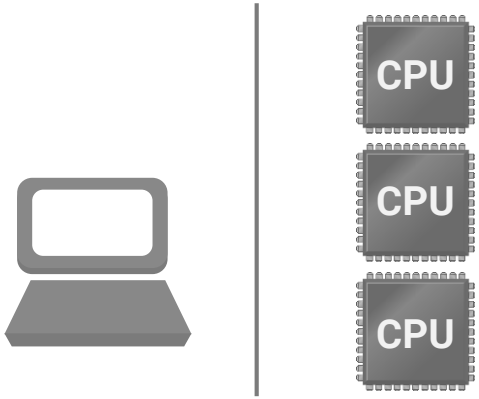
- Technischer Hintergrund: Parallele Architekturen
- Parallele Herausforderungen
  - Deadlocks!
  - Race Conditions!
- Synchronisationsprimitive
- Architekturmuster
  - Traditionelle Client-Server Architektur
  - Moderne „Asynchrone“ Programmierung

# Parallele Architekturen



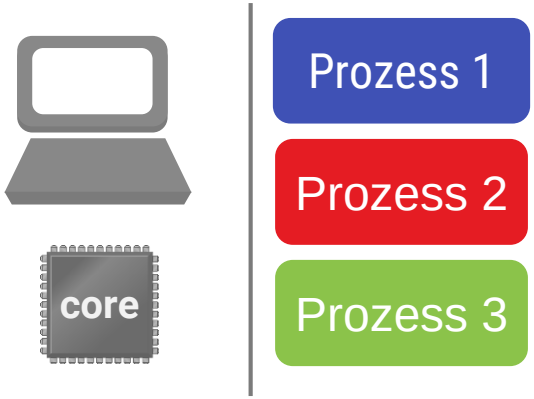
Distributed System („verteilt“)

Mehrere Rechner



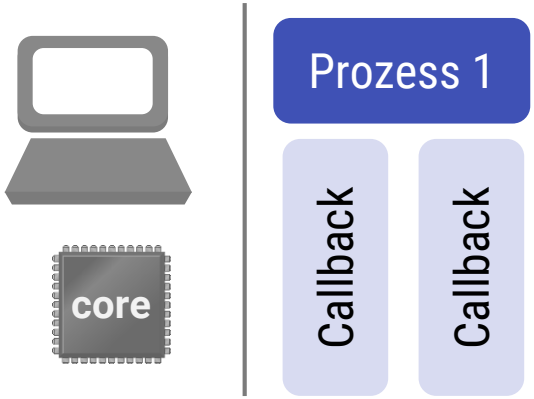

Shared-Memory System

Ein Rechner, mehrere CPUs




Präemptives Multi-Tasking

Eine CPU, mehrere Tasks




Kooperatives Multi-Taksing

Eine CPUs, ein Task

# Wie funktioniert es?

## Verteiltes System

- Netzwerk (Ethernet, Fiberchannel, InfiniBand, etc.)
- Kommunikation:
  - Nachrichten schicken („message passing“)

## Shared Memory

- Gemeinsamer Hauptspeicher
  - I.d.R. Cache-koheränt (keine explizite Synchronisation nötig)
- Kommunikation
  - Gemeinsamen Speicher lesen/schreiben
  - Semaphoren / mutexes
  - Nachrichten schicken (simuliert in Software)

# Wie funktioniert es?

## Präemptives Multi-Tasking

- Automatisches Umschalten
  - Typisch: alle 20-50ms (20-50x pro Sekunde)
  - Keine echte Parallelverarbeitung
  - Aber es sieht für den Nutzer so aus
- Zwei Modelle für Multi-Tasking
  - „Multi-Threading“: gemeinsamer Hauptspeicher
  - Mehrere Prozesse: getrennter Speicher (simuliert)
    - Jeder Prozess enthält i.d.R. mehrere Threads
- Kommunikation
  - Gemeinsamen Speicher lesen/schreiben
  - Semaphoren / mutexes
  - Nachrichten schicken (simuliert in Software)

# Wie funktioniert es?

## Kooperatives Multi-Tasking

- Explizit programmiertes Umschalten
  - Wenn man Code schreibt, muss man ihn selbst in Häppchen aufteilen und umgschalten
- Architektur
  - Callbacks
  - Event-driven programming
  - Genau wie bei den meisten GUIs! (→ Eventqueues)
- Kommunikation:
  - Gemeinsamen Speicher lesen/schreiben
  - Callbacks / Events „posten“
    - Neues Ereignis in Ereigniswarteschlange

# Multi-Tasking: Begriffe

## Präemptiv / Multi-Core / Multi-CPU

- Nebenläufiger Kontrollfluss (Programm)
- Task: Oberbegriff
  - Threads: gemeinsamer Adressraum
  - Prozesse: getrennter Adressraum
- Tasks werden vom OS automatisch auf die verfügbaren CPUs/Cores verteilt

# Parallele Herausforderungen

## Nebenläufigkeit ist schwierig

- Parallele Algorithmen
  - Nicht jedes Problem kann man parallel lösen
  - Unterschiedlich schwer (Vorlesung Parallele Algorithmen)
- Synchronisationsprobleme
  - „Race-Conditions“ – inkonsistenter paralleler Datenzugriff
  - „Deadlocks“ – zyklisches Warten

# Beispiel: Drucker-Spooler

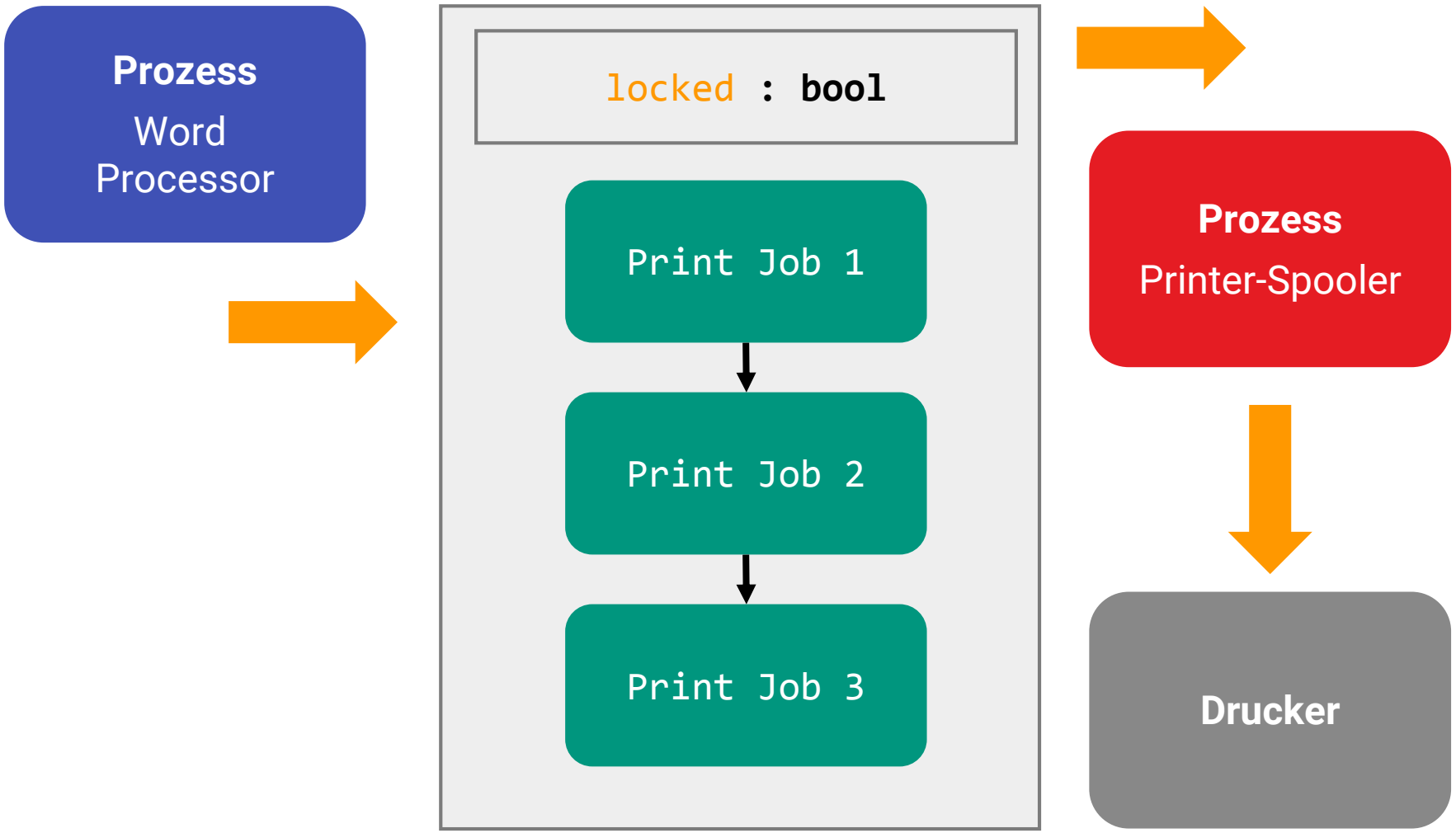
## Zwei Prozesse

- Einer produziert Dokumente
- Ein zweiter schickt die zum Drucker
  - Man kann weiterarbeiten, während gedruckt wird

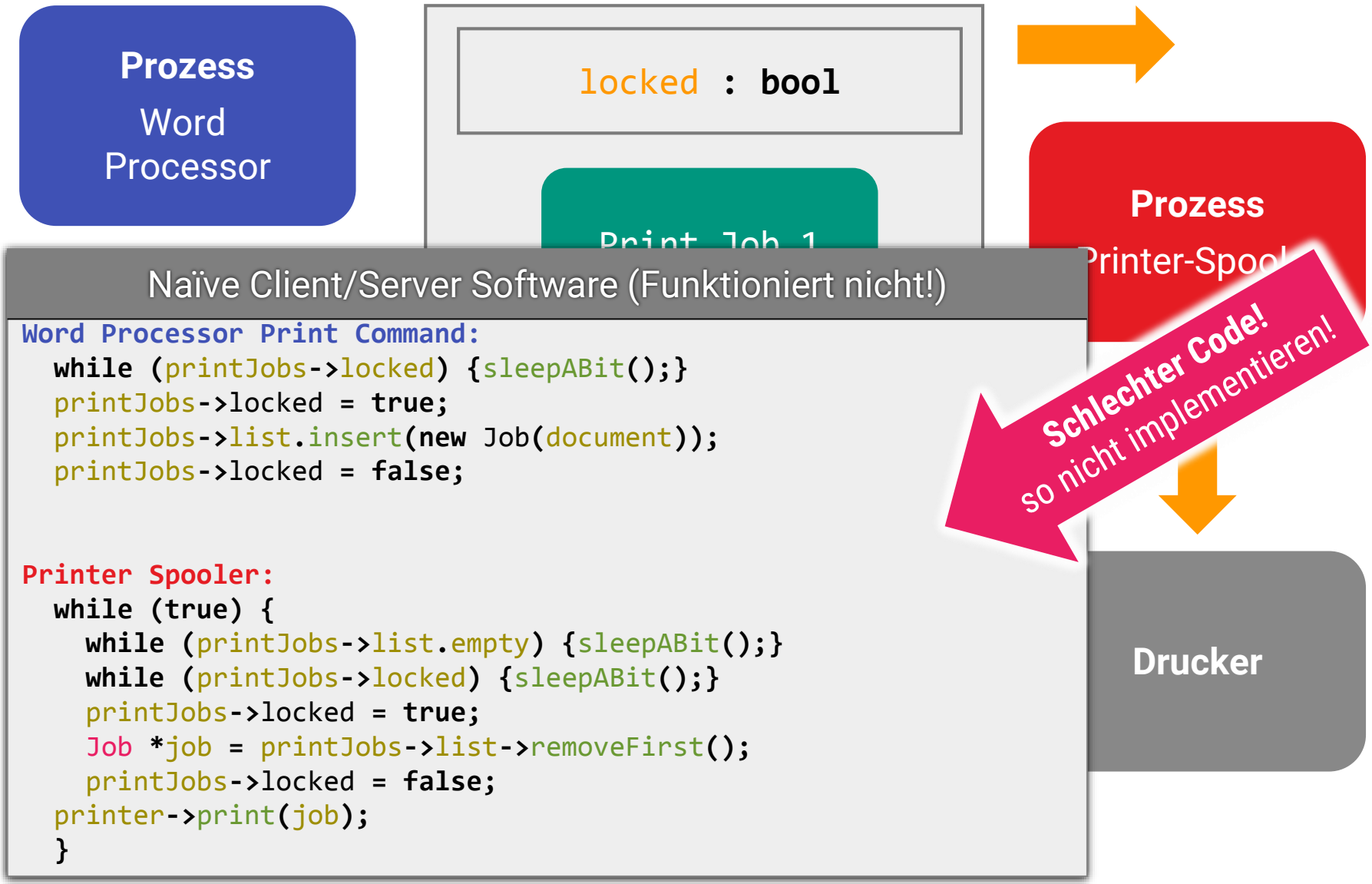
## Implementation

- Verkettete Liste von Druckerjobs
- Gemeinsamer Speicher (einfachste Situation)

# Race Conditions



# Race Conditions




Naïve Client/Server Software (Funktioniert nicht!)

```
Word Processor Print Command:
  while (printJobs->locked) {sleepABit();}
  printJobs->locked = true;
  printJobs->list.insert(new Job(document));
  printJobs->locked = false;


Printer Spooler:
  while (true) {
    while (printJobs->list.empty) {sleepABit();}
    while (printJobs->locked) {sleepABit();}
    printJobs->locked = true;
    Job *job = printJobs->list->removeFirst();
    printJobs->locked = false;
  printer->print(job);
  }
```

# Race Conditions

## Problem: Reihenfolge der Zugriffe

- Inkonsistente Zustände möglich
- Abfrage und Schreiben von Variable „`locked`“ kann unterbrochen werden

## Lösung

- Atomare Operation „`getLockIfPossible()`“
- Atomare Operation „`releaseLock()`“
- Atomar: Während Ausführung kein Zugriff durch nebenläufigen Vorgang
- Hardwareunterstützung (alle moderenen CPUs)
  - Softwarelösung möglich (eher akademisch, Peterson‘s solution)

# Synchronisationsprimitive

## Fertige Komponenten (moderne OS)

- Mutex – atomare „Lock“
- Semaphore – atomare Zähler
  - Warten auf „mindestens 1 Dokument im Druckerspooler“
- Shared Memory
  - Zwischen mehreren „threads“: gesamter Speicher
  - Zwischen „Prozessen“: Muss vom OS angefordert werden
- Message Queues
  - Netzwerk (z.B. tcp/ip ports)
  - Simulation von Nachrichten auch bei shared memory

**Bei allen:** Warten kostet **keine** Rechenzeit (Task schläft)

# Beispiel: C++ und QT

## Prozesse und Threads

- `QThread`
  - Abstrakte Methode
    ```
    protected: virtual int exec() = 0;
    ```
    enhält Code für Thread
  - Ableiten, um neue Thread-Objekte zu definieren
  - Methode `run()` startet Thread (von außen)
- `QProcess`
  - Kapselt neuen Prozess
  - Benötigt Dateinamen eines ausführbaren Programms (Windows Prozess Model ist an Dateien gebunden)
  - Message-Passing über Standard-Ein/Ausgabe möglich

# Beispiel: C++ und QT

## Synchronisationsprimitive in QT

- `QCriticalSection`
  - Methoden `lock()`, `unlock()`
- `QSemaphore`
  - Methoden `acquire()`, `release()`, `available()`
- Traditionelle message queues: `QLocalSocket`
- QT-Message queues mit Event-Modell
  - Abgebildet auf **`signals`** + **`slots`** (event-driven)
  - `QThread` kann **`signale`** + **`slots`** senden/empfangen
  - Jeder Thread hat eine eigene Event-Queue
  - Nur ein GUI-Thread erlaubt (Widgets sind nicht „thread-safe“)

# Verteilte Systeme

## Kommunikation zwischen verschiedenen Rechnern

- Internet Protokoll (TCP/IP)
  - Jeder Rechner hat eine IP-Adresse (z.B. „134.93.178.2“)
  - Jeder Rechner kann bis zu $2^{16}$ (64K) Ports haben
  - Bidirektionale Punkt-zu-Punkt Verbindung (IP+Port zu IP+Port)
- Kommunikation zwischen verschiedenen oder dem auch auf dem gleichen Rechner
  - Über Prozess- und Rechnergrenzen hinweg
- BSD-Sockets
  - Standard Softwareabstraktion, UNIX+Windows
  - Gekapselt in Klasse `QTcpSocket`

# Verteilte Systeme

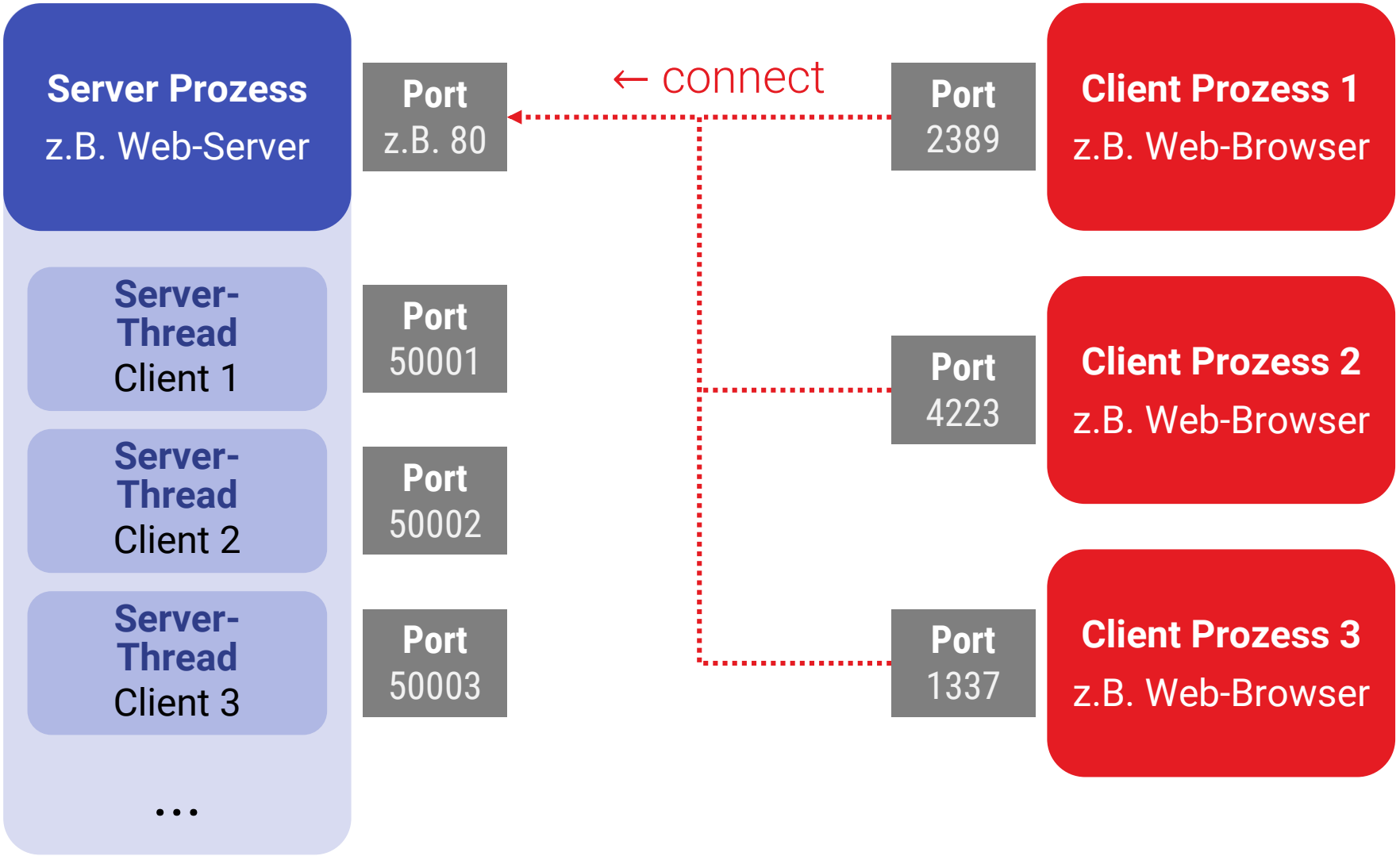
## Server Kommandos

- `socket()` – erzeugt eine Socket
- `bind()` – setzt Parameter (Port, Protokoll, etc.)
- `listen()` – wartet auf Kontakt von außen (Task schläft)
- `accept()` – Verbindung akzeptiert: erzeugt neuen Socket!
- `read()`/`write()` – schickt/empfängt Daten (Binär)

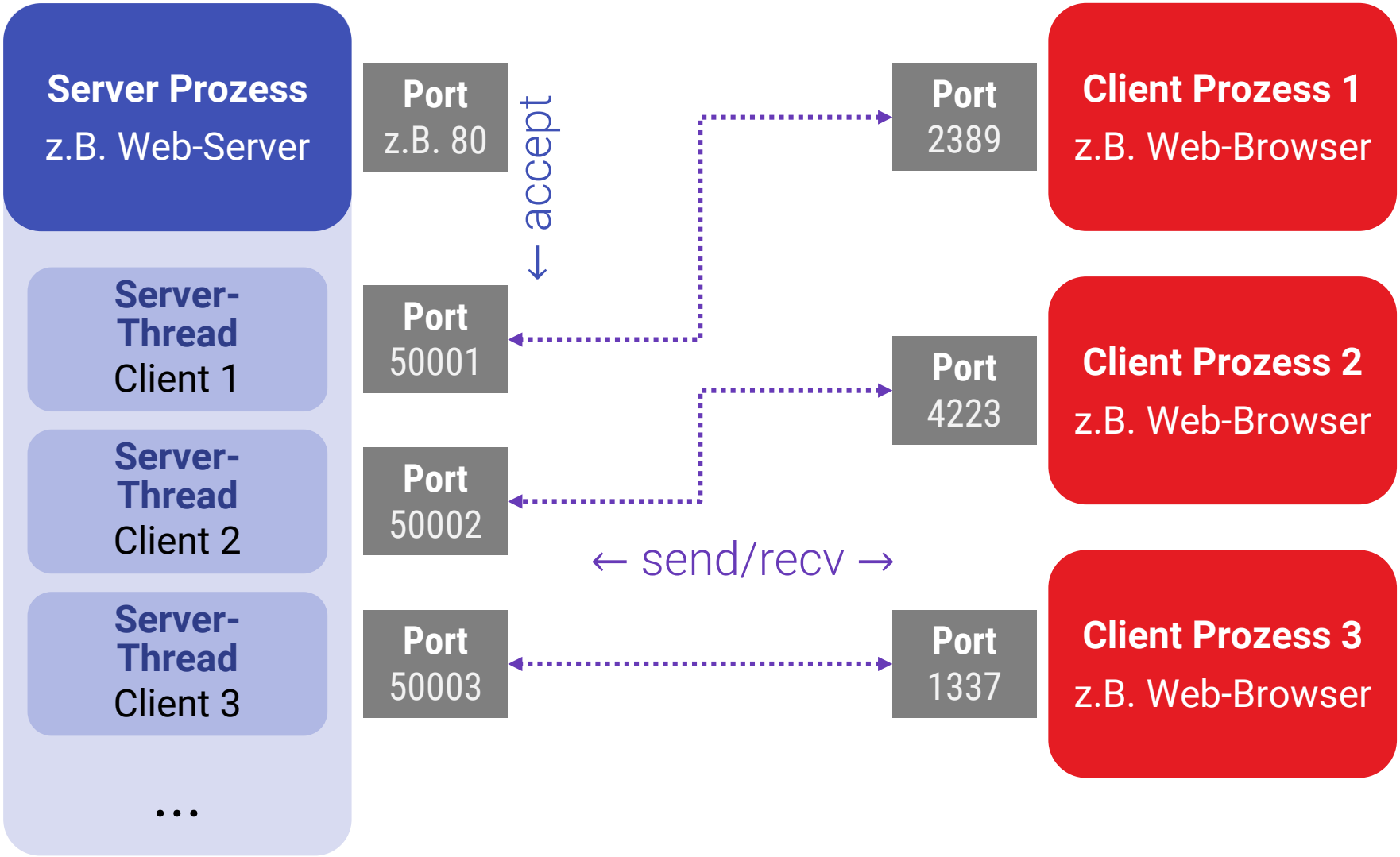
## Server Kommandos

- `socket()` – erzeugt eine Socket
- `connect()` – Verbindet mit Server-Socket (setzt auch Par.)
- `read()`/`write()` – schickt/empfängt Daten (Binär)
- `close()` – schließt Verbindung (auch für Server)

# Server Architektur



# Server Architektur



# Standard Client/Server Architektur

**Server-Hauptprozess (Pseudo-Code!)**

```
while (true) { // server runs forever
    socket = Warte auf Verbindung;
    Erzeuge neuen Thread für socket;
    Starte Thread (Parameter: Socket,
                   Sever-Daten);
}
```

**Server-Thread (Pseudo-Code!)**

```
while (true) {
    Warte auf Kommando(socket)
    falls commando-1:
      Empfange weitere Parameter;
      Hole Daten;
      Sende Daten an Client;
    falls commando-2:
      Ähliche Komunikation mit client;
    ...
    falls commando-ende: break;
}
Schließe Verbindung;
```

**Server (Pseudo-Code!)**

```
verbinde mit Server;

Schicke Kommando;
Schicke Paramter;
Schicke / Empfange Daten;

Schicke Kommando;
Schicke Paramter;
Schicke / Empfange Daten;

...

Schicke Ende-Commando;

Schließe Verbindung;
```

# Standard Client/Server Architektur

## Server-Hauptprozess (Pseudo-Code!)

```
Socket *sock
   = new Socket(80, TYPE_TCP_IP);
while (true) { // server runs forever
   sock.listen();
   Socket *clSock = sock.accept();
   ServerThread *newClientServ
      = new ServerThread(clSock, this);
   newClientServ->run();
}
```

## Server-Thread (Pseudo-Code!)

```
while (true) {

   sock->read(command);

   if (command==first) {send first}
   if (command==second) {send second}
   if (command==end) {break;}

}

sock->close();
```

## Client (Pseudo-Code!)

```
Socket *sock = new Socket();
sock->connect(
  80,
  nslookup("www.myserver.de"),
  TYPE_TCP_IP
);
sock->write(
  "GET /index.html HTTP/1.1\n“
  "Host: www.myserver.de\n");
sock->read(buffer);
displayPage(buffer)
```

# Standard Client/Server-Architekur

## Serverprozess

- Wartet auf Verbindungen
- Erzeugt (sofort) einen neuen Thread für jede Verbindung
- Jeder Thread bedient einen Client

## Motivation

- Hohe Latenzen
  - Internet: 10ms–100ms realistisch
  - Klassische Festplatte: 10-20ms pro Zugriff
- Umschalten auf andere Threads um CPU auszunutzen

# Es ist 2020…

## Neue Welt

- Latenzen dramatisch gesunken
  - Schnelle I/O-Geräte
  - Schnelle (lokale) Netzwerke
- Tausende Verbindungen (Internet-Server)
  - Threads brauchen einige Ressourcen
  - Kann ein Bottleneck werden

# Es ist 2020…

**Latenzen** (IOPS = I/O Operationen/sec)

7200rpm HD: ca. 75-100 IOPS (10ms)

Consumer SSD: 100.000 IOPS (10µs)

Profi SSD-Array: bis zu 10.000.000 IOPS (100ns)

InifiniBand: Latenz ca. 1µs

Linux-Thread-Switch/IPC: ca. 1-10µs (ca. 10000 Takte)

- Experimente im Netz: Nachrichten via „Pipe“, Sockets, Semaphoren brauchen ca. 3-4µsec auf aktuellem Core-ix
- „Busy Waiting / Polling“ ca. 150nsec

# Moderne Architektur

## Moderne Server

- „Non-Blocking-IO“
  - Kein Warten auf Daten
  - Direkt nächsten Job anfangen
  - Alles in einem Thread
- Jeder Thread arbeitet mit vielen Clients
  - Event-Queues in jeden Thread
  - Ereignisse, falls neue Daten verfügbar werden

## Beispiele

- Node.JS, Python `asyncio`, Boost `asio` (C++)

# Zusammenfassung

## Client-Server Architektur

- Nachrichtenaustausch zwischen Client und Server
- Typischerweise Ereignis-orientierte Architektur in beiden
- Latenzen verstecken / multi-Core ausnutzen via Threading:
  - Server erzeugt neuen Thread für jede Anfrage
  - Thread „blocked“ bis Daten verfügbar (einfach zu prog.)
  - Schreiben auf „Datenbank“ via Mutex/Semaphoren gesichert
- Moderne Serverarchitekturen nutzen „non-blocking“-I/O
  - Effizienter bei niedrigen Latenzen bzw. sehr vielen Anfragen

# Architekturmuster

## Nur einige wenige Muster

- Klassische Client/Server-Architektur:
  - Threads erzeugen für Verbindungen
- Event-driven:
  - Ereignisse in Callbacks verwandeln
  - Verteilen an Komponentenbaum
  - Klassisches OOP-Design:
    - Eine Art von abstrakter „`processEvent`“-Methode
  - „Funktional“
    - Datenflussgraph (evtl. zyklisch = rekursiv)
  - Reactive programming
    - Events in Datenflussgraph

# Moderner Server