

# Betriebssysteme und nebenläufige Programmierung

SoSe 2025

Roland Wismüller  
Betriebssysteme / verteilte Systeme  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. April 2025

# Betriebssysteme und nebenläufige Programmierung

SoSe 2025

## 4 Kommunikation



## Inhalt:

- ➔ Einführung
- ➔ Elementare Kommunikationsmodelle
- ➔ Adressierung
- ➔ Varianten und Erweiterungen
  
- ➔ Tanenbaum 2.3.8, 8.2.3, 8.2.4
- ➔ Stallings 5.6, 6.7, 13.3
- ➔ Nehmer/Sturm 7

### Methoden zur Kommunikation

- ➔ Speicherbasierte Kommunikation
  - ➔ über gemeinsamen Speicher (s. Erzeuger/Verbraucher-Problem)
  - ➔ i.d.R. zwischen Threads desselben Prozesses
  - ➔ gemeinsamer Speicher auch zwischen Prozessen möglich
  - ➔ Synchronisation muß explizit programmiert werden
  
- ➔ Nachrichtenbasierte Kommunikation
  - ➔ Senden / Empfangen von Nachrichten (über das BS)
  - ➔ i.d.R. zwischen Threads verschiedener Prozesse
  - ➔ auch über Rechnergrenzen hinweg möglich
  - ➔ implizite Synchronisation



### Nachrichtenbasierte Kommunikation

- ➔ Nachrichtenaustausch durch zwei Primitive:
  - ➔ `send(Ziel, Nachricht)` – Versenden einer Nachricht
  - ➔ `receive(Quelle, Nachricht)` – Empfang einer Nachricht
    - ➔ oft: spezieller Parameterwert für beliebige Quelle  
evtl. Quelle auch als Rückgabewert
  
- ➔ Implizite Synchronisation:
  - ➔ Empfang einer Nachricht erst **nach** dem Senden möglich
  - ➔ `receive` blockiert, bis Nachricht vorhanden ist
    - ➔ manchmal zusätzlich auch nichtblockierende `receive`-Operationen; ermöglicht *Polling*



### Beispiel: Erzeuger/Verbraucher-Kommunikation mit Nachrichten

- ➔ Typisch: BS puffert Nachrichten bis zum Empfang
  - ➔ Puffergröße wird vom BS bestimmt (meist konfigurierbar)
  - ➔ falls Puffer voll ist: Sender wird in `send()`-Operation blockiert (Flußkontrolle,  Rechnernetze I)

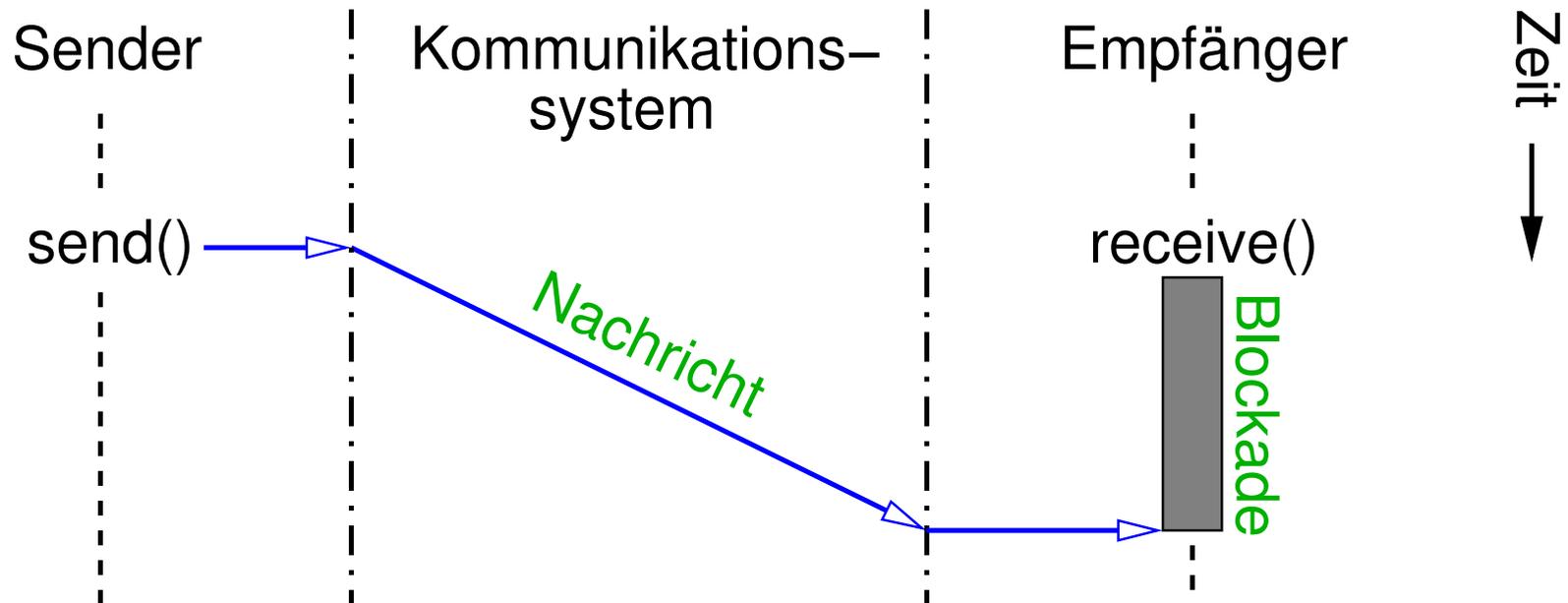
```
void producer() {
    int item;
    Message m;
    while (true) {
        item = produceItem() ;
        build_message(&m, item);
        send(consumer, m);
    }
}
```

```
void consumer() {
    int item;
    Message m;
    while(true) {
        receive(producer, &m);
        item = extractItem(m);
        consumeItem(item);
    }
}
```

### Klassifikation (nach Nehmer/Sturm)

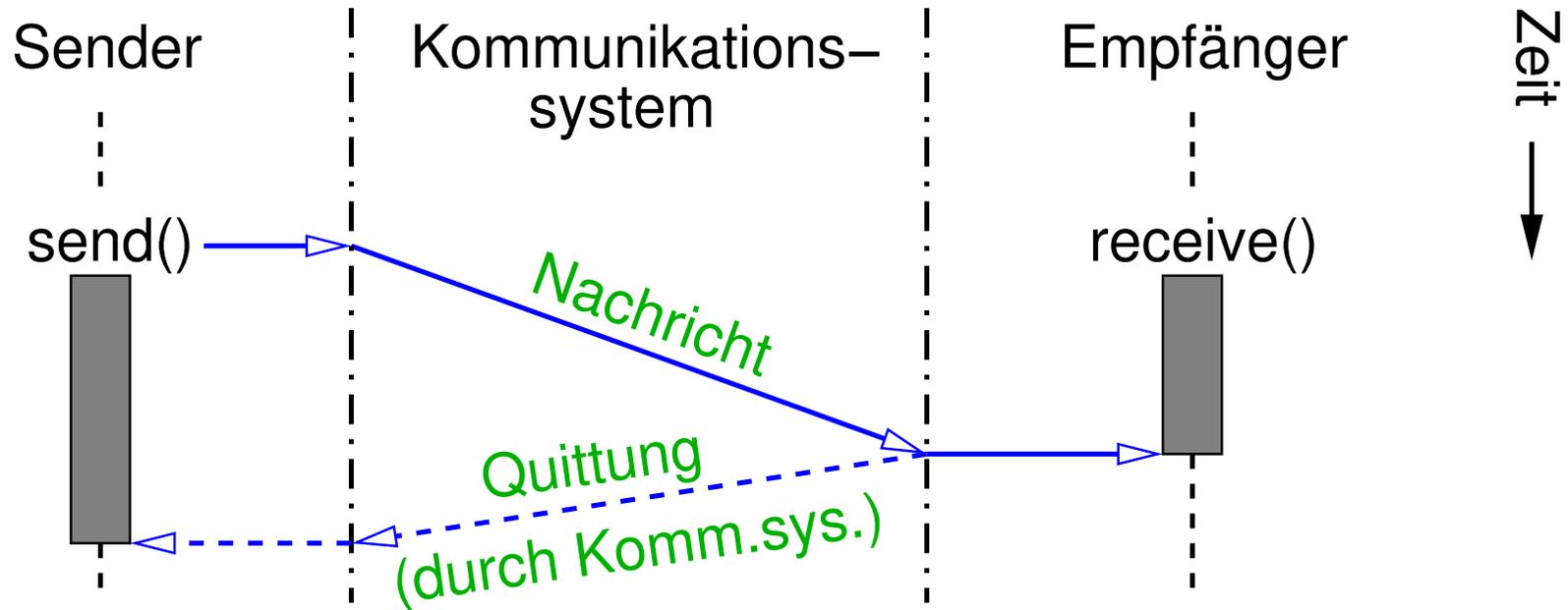
- ➔ Zeitliche Kopplung der Kommunikationspartner:
  - ➔ synchrone vs. asynchrone Kommunikation
  - ➔ auch: blockierende vs. nicht-blockierende Kommunikation
  - ➔ wird der Sender blockiert, bis der Empfänger die Nachricht empfangen hat?
  
- ➔ Muster des Informationsflusses:
  - ➔ Meldung vs. Auftrag
  - ➔ Einweg-Nachricht oder Auftragserteilung mit Ergebnis?

### Asynchrone Meldung



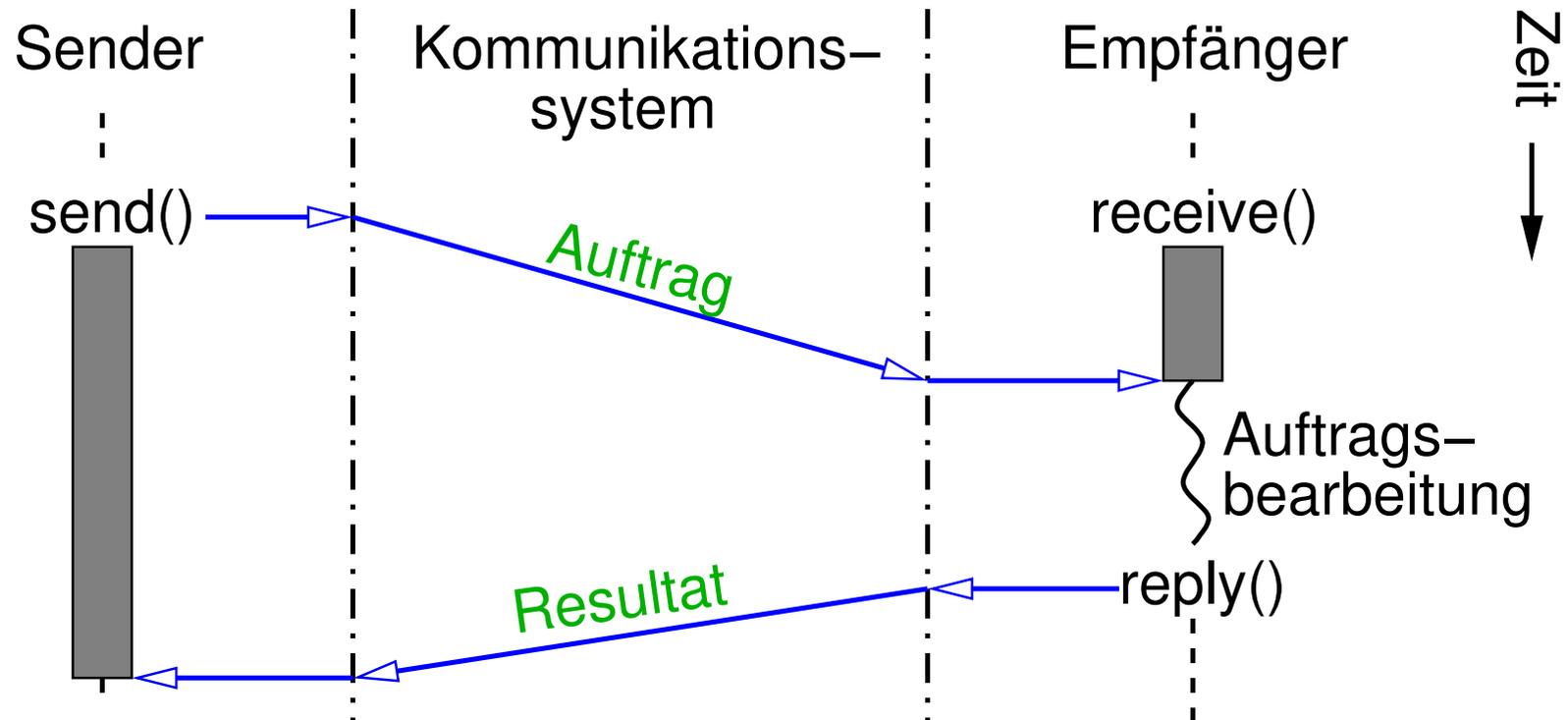
- ➔ Sender wird nicht mit Empfänger synchronisiert
- ➔ Kommunikationssystem (BS) muß Nachricht ggf. puffern, falls Empfänger (noch) nicht auf Nachricht wartet

### Synchrone Meldung



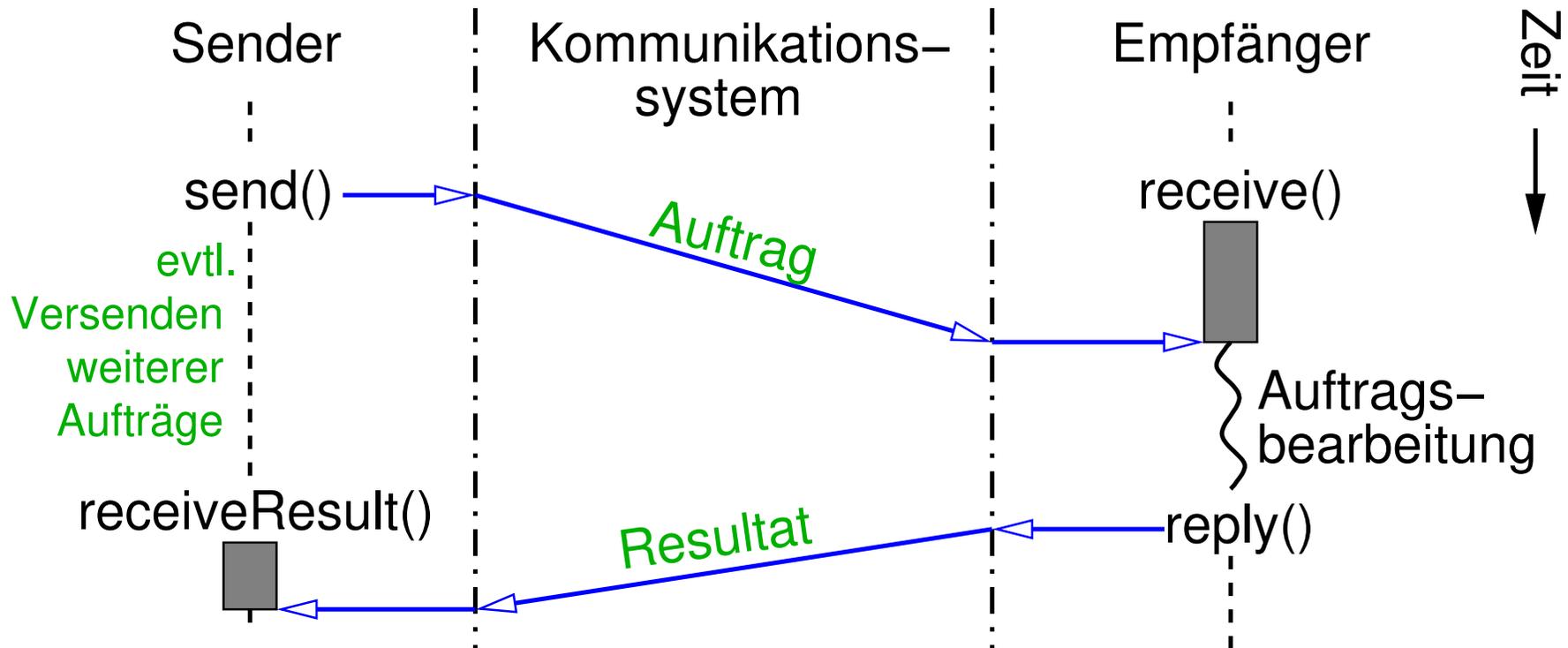
- ➔ Sender wird blockiert, bis Nachricht empfangen ist
- ➔ **Rendezvous**-Technik
- ➔ Keine Pufferung erforderlich

### Synchroner Auftrag



- ➔ Empfänger sendet Resultat zurück
- ➔ Sender wird blockiert, bis Resultat vorliegt

### Asynchroner Auftrag



- ➔ Sender kann mehrere Aufträge gleichzeitig erteilen
- ➔ Parallelverarbeitung möglich

### Wie werden Sender bzw. Empfänger festgelegt?

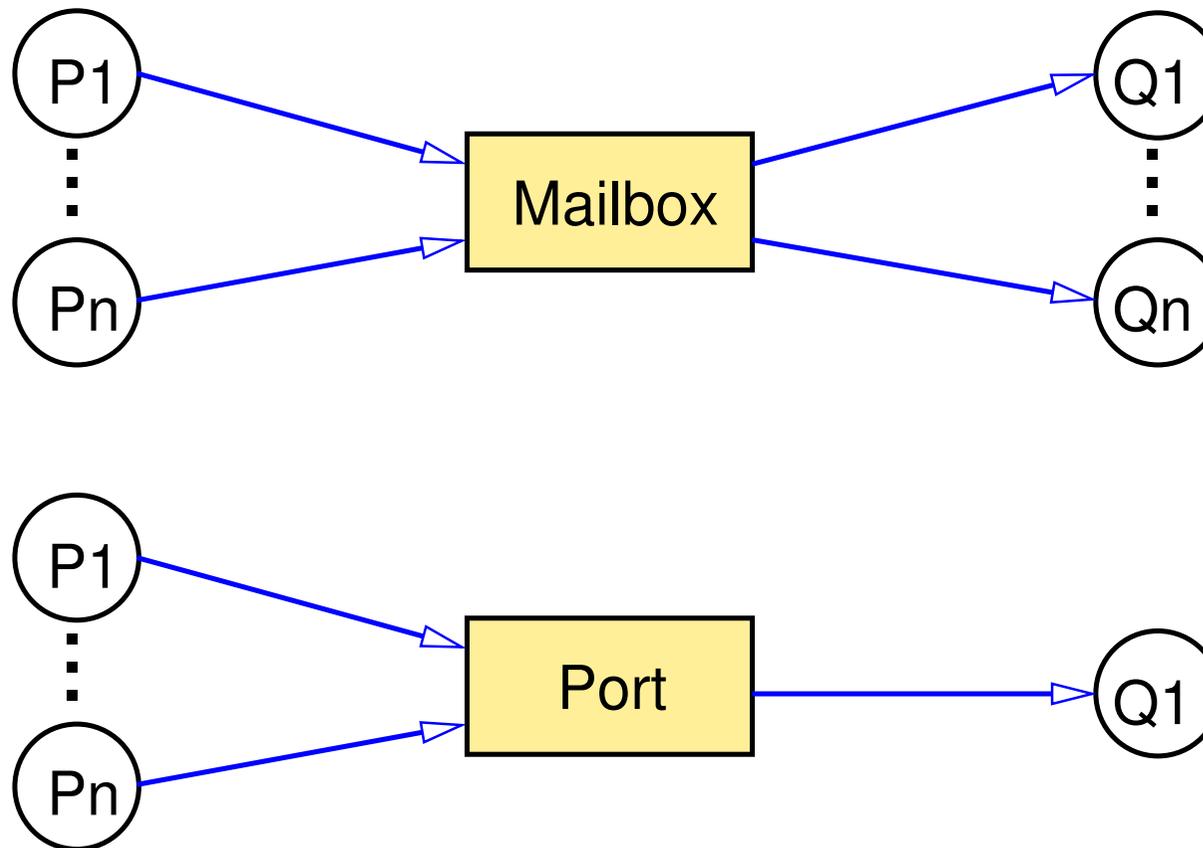
- ➔ Direkte Adressierung
  - ➔ Kennung des jeweiligen Prozesses
- ➔ Indirekte Adressierung
  - ➔ Nachrichten werden an Warteschlangen-Objekt (**Mailbox**) gesendet und von dort gelesen
  - ➔ Vorteil: höhere Flexibilität
    - ➔ Mailbox kann von mehreren Prozessen gelesen werden
      - ➔ **Port**: Mailbox mit nur einem möglichen Empfänger-Prozess
    - ➔ Empfänger kann mehrere Mailboxen / Ports besitzen

(Anm.: Für Sender und Empfänger werden nur Prozesse betrachtet)

### Mailbox und Port

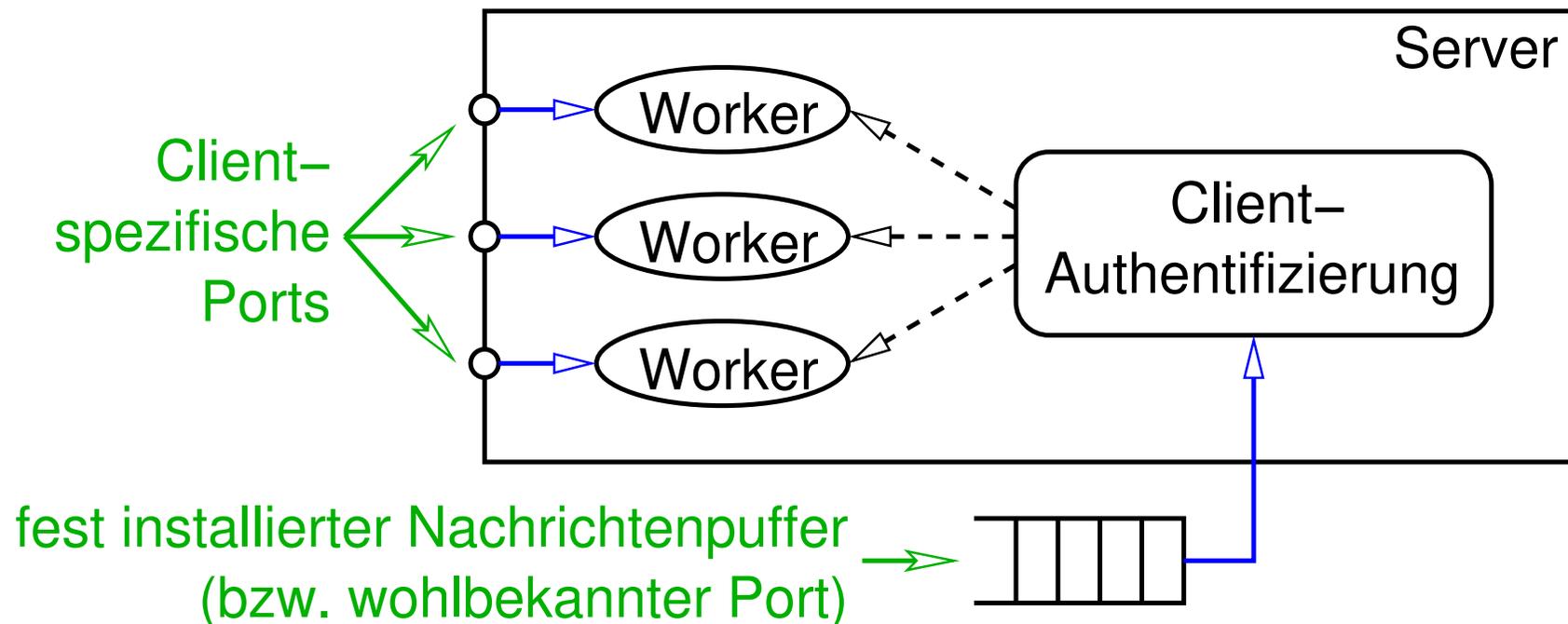
Sendende Prozesse

Empfangende Prozesse



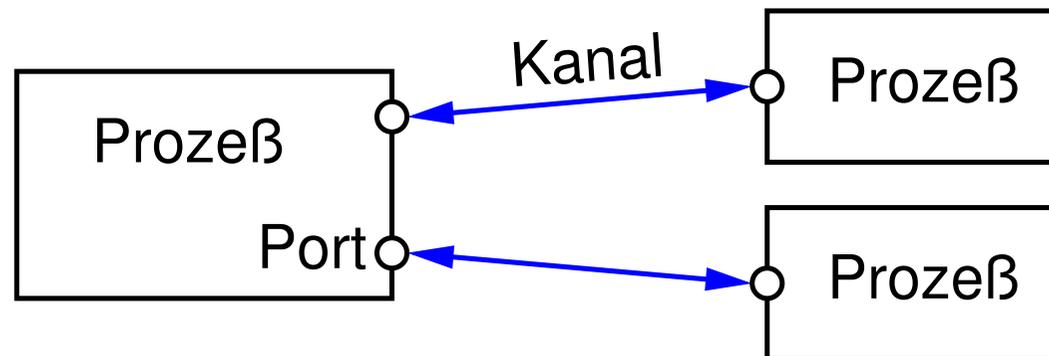
### Anwendung von Ports: z.B. selektiver Nachrichteneingang

- ➔ Server kann nach Anmeldung eines Clients für jeden Client einen eigenen Port erzeugen
- ➔ jeder Port kann durch einen eigenen Worker-Prozeß (oder Thread) bedient werden



### Kanäle

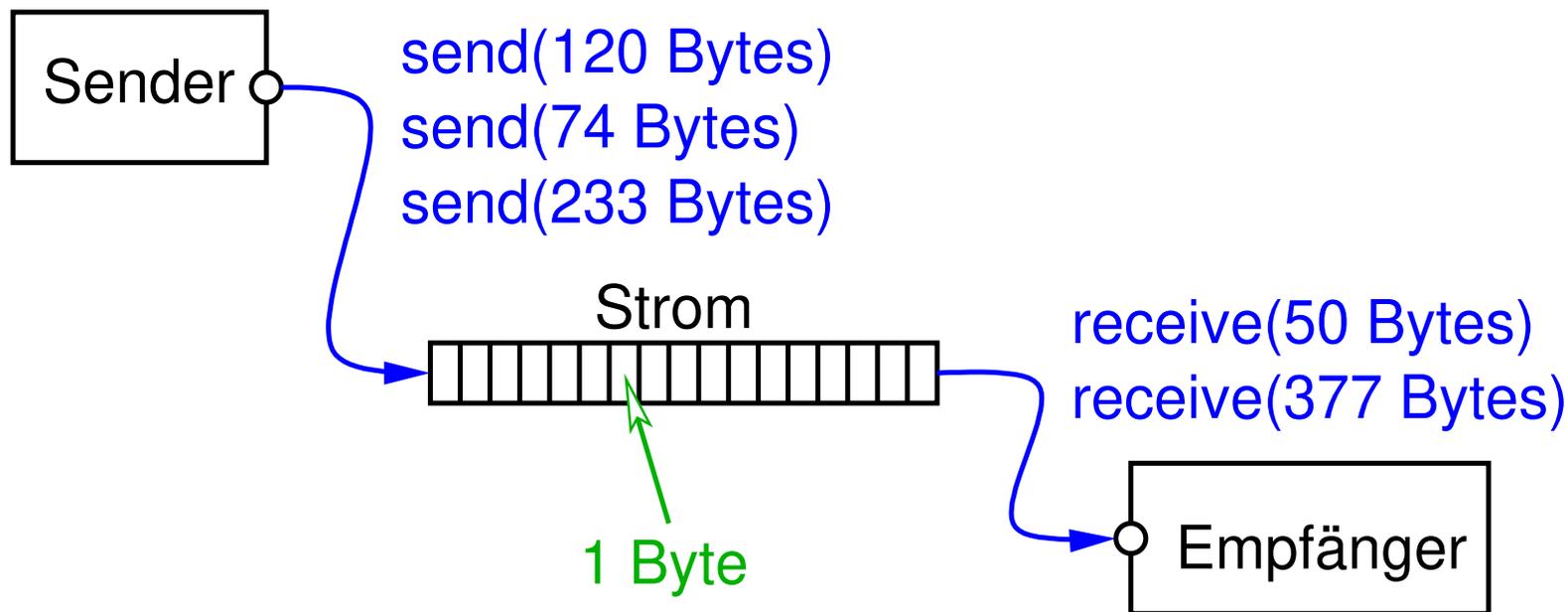
- ➔ Bisher: verbindungslose Kommunikation
  - ➔ wer Adresse eines Ports kennt, kann senden
- ➔ Kanal: logische Verbindung zw. zwei Kommunikationspartnern



- ➔ expliziter Auf- und Abbau einer Verbindung (zw. zwei Ports)
- ➔ meist bidirektional: Ports für Senden und Empfangen
- ➔ garantierte Nachrichtenreihenfolge (FIFO)
- ➔ Beispiel: TCP/IP-Verbindung

### Ströme (*Streams*)

- ➔ Kanal zur Übertragung von Sequenzen von Zeichen (Bytes)
- ➔ Keine Nachrichtengrenzen oder Längenbeschränkungen
- ➔ Beispiele: TCP/IP-Verbindung, UNIX *Pipes*, Java *Streams*





### Ströme in POSIX: *Pipes*

- ➔ *Pipe*: Unidirektionaler Strom
- ➔ Schreiben von Daten in die Pipe:
  - ➔ `write(int pipe_desc, char *msg, int msg_len)`
    - ➔ `pipe_desc`: Dateideskriptor
    - ➔ bei vollem Puffer wird Schreiber blockiert
- ➔ Lesen aus der Pipe:
  - ➔ `int read(int pipe_desc, char *buff, int max_len)`
    - ➔ `max_len`: Größe des Empfangspuffers `buff`
    - ➔ Rückgabewert: Länge der tatsächlich gelesenen Daten
  - ➔ bei leerem Puffer wird Leser blockiert



### Ströme in POSIX: Erzeugen einer *Pipe*

➔ Unbenannte (*unnamed*) *Pipe*:

➔ ist zunächst nur im erzeugenden Prozeß bekannt

➔ Dateideskriptoren werden an Kindprozesse vererbt

➔ Beispielcode:

```
int pipe_ends [2]; // Dateideskriptoren der Pipe-Enden
pipe(pipe_ends); // Erzeugen der Pipe
if (fork() != 0) {
    // Vaterprozeß
    write(pipe_ends [1], data, ...);
} else {
    // Kindprozeß (erbt Dateideskriptoren)
    read(pipe_ends [0], data, ...);
}
```

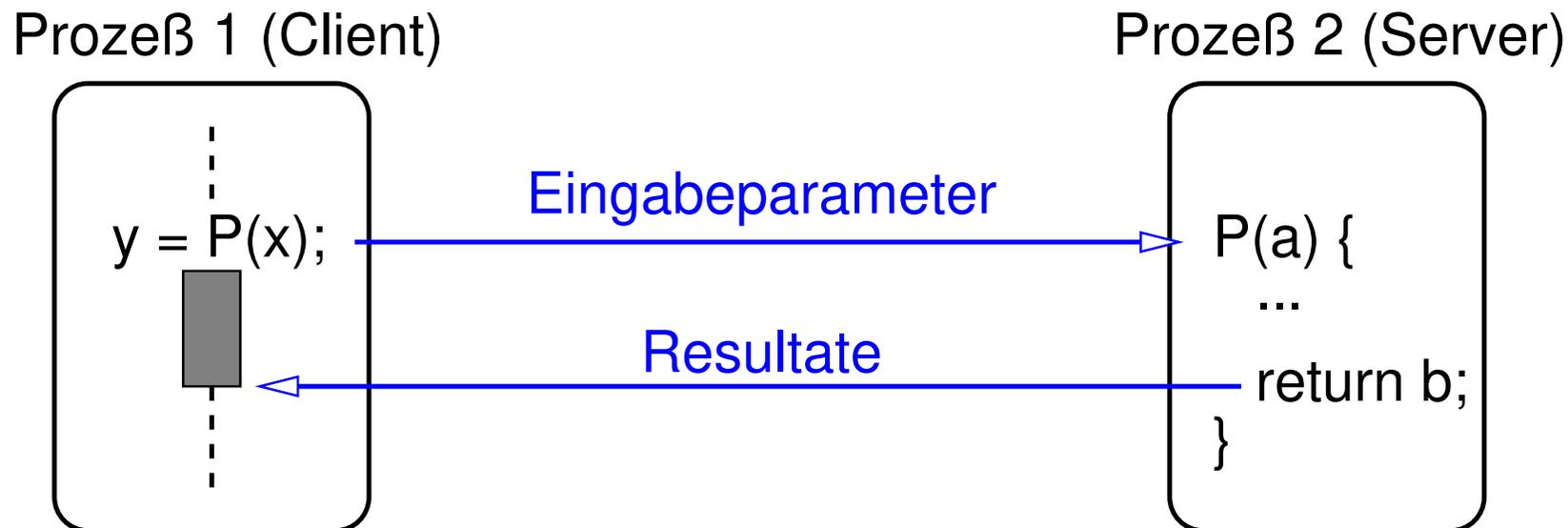


### Ströme in POSIX: Erzeugen einer *Pipe* ...

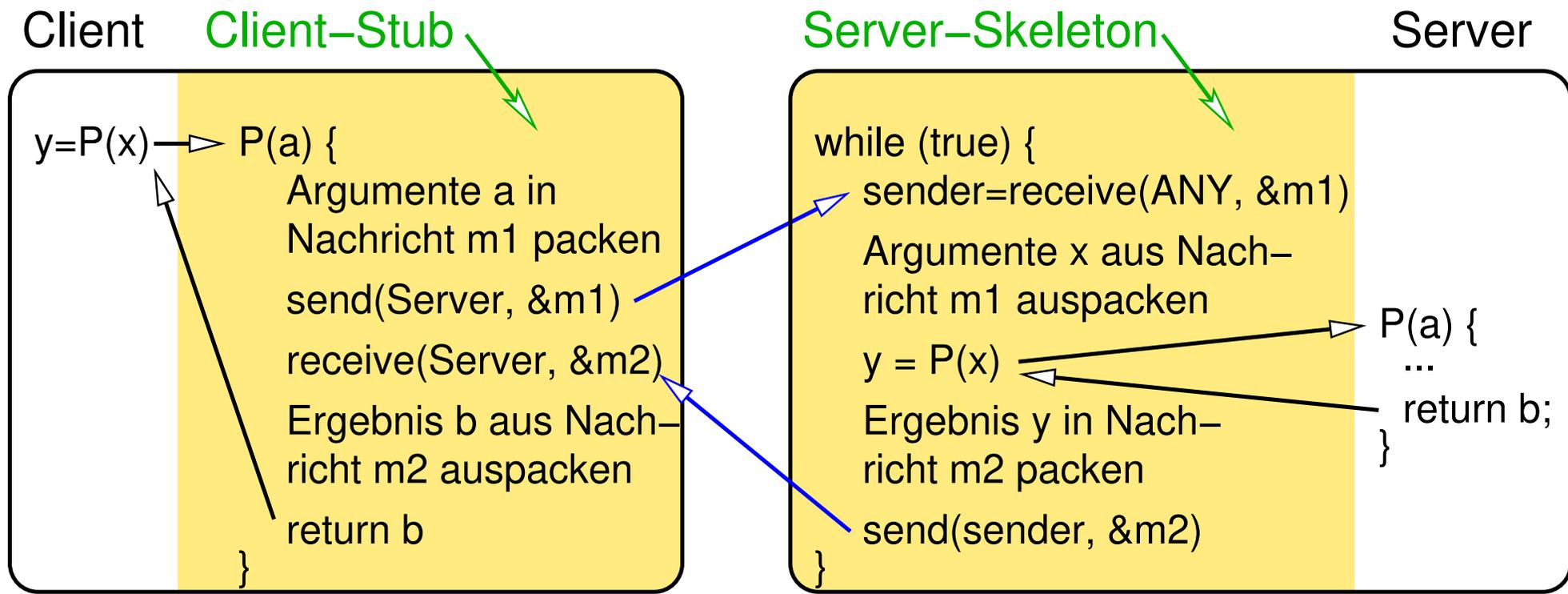
- ➔ Benannte (*named*) *Pipe*:
  - ➔ ist als spezielle „Datei“ im Dateisystem sichtbar
  - ➔ Zugriff erfolgt exakt wie auf normale Datei:
    - ➔ Systemaufrufe `open`, `close` zum Öffnen und Schließen
    - ➔ Zugriff über `read` und `write`
- ➔ Erzeugung:
  - ➔ Systemaufruf `mkfifo(char *name, mode_t mode)`
  - ➔ in Linux auch Shell-Kommando: `mkfifo <name>`
- ➔ Löschen durch normale Dateisystem-Operationen

### Remote Procedure Call (RPC)

- ➔ Idee: Vereinfachung der Realisierung von synchronen Aufträgen
- ➔ RPC: Aufruf einer Prozedur (Methode) in einem anderen Prozeß



## Realisierung eines RPC



➔ Client-Stub und Server-Skeleton werden i.d.R. aus Schnittstellenbeschreibung generiert: **RPC-Compiler**



### RPC: Diskussion

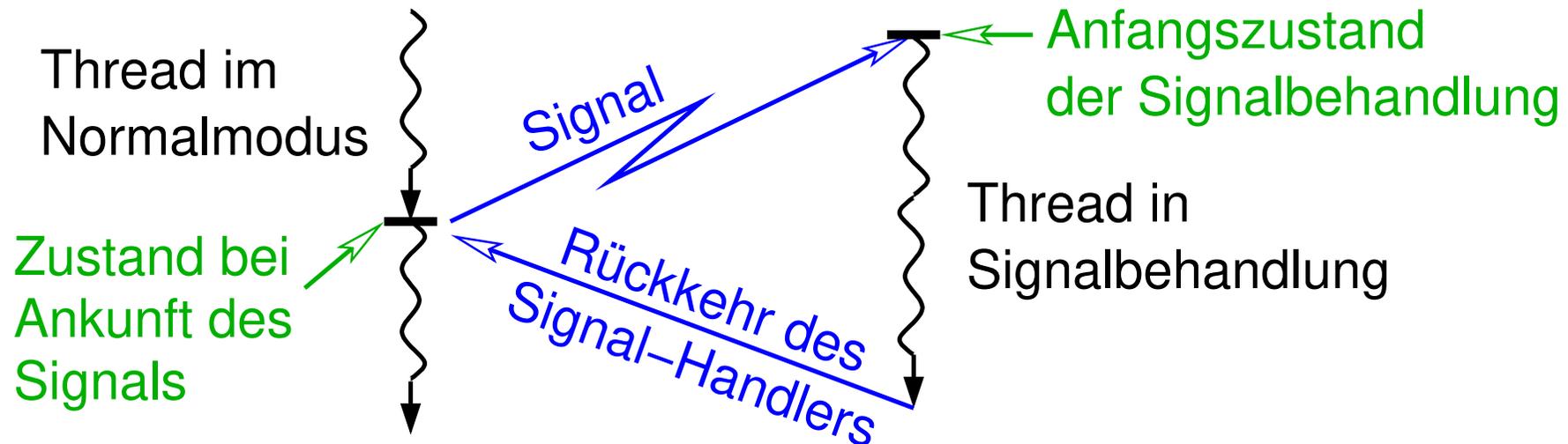
- ➔ Client muß sich zunächst an den richtigen Server binden
  - ➔ Namensdienste verwalten Adressen der Server
- ➔ Danach: RPC syntaktisch exakt wie lokaler Prozeduraufruf
- ➔ Semantische Probleme:
  - ➔ *Call-by-Reference* Parameterübergabe sehr problematisch
  - ➔ Kein Zugriff auf globale Variablen möglich
  - ➔ Nur streng getypte Schnittstellen verwendbar
    - ➔ Datentyp muß zum Ein- und Auspacken (***Marshaling***) genau bekannt sein
  - ➔ Behandlung von Fehlern in der Kommunikation?
- ➔ Beispiel: Java RMI



### Signale

- ➔ Erlauben sofortige Reaktion des Empfängers auf eine Nachricht (asynchrones Empfangen)
- ➔ Asynchrone Unterbrechung des Empfänger-Prozesses
  - ➔ „Software-Interrupt“: BS-Abstraktion für Interrupts
- ➔ Operationen (abstrakt):
  - ➔ `Receive(Signalnummer, HandlerAdresse)`
  - ➔ `Signal(Empfänger, Signalnummer, Parameter)`
- ➔ Ideal: Erzeugung eines neuen Threads beim Eintreffen des Signals, der Signal-Handler abarbeitet
- ➔ Historisch: Signal wird durch unterbrochenen Thread behandelt

### Behandlung eines Signals im unterbrochenen Thread



- ➔ BS sichert Threadzustand im Thread-Kontrollblock und stellt Zustand für Signalbehandlung her
- ➔ Kein wechselseitiger Ausschluß (z.B. Semaphore) möglich
  - ➔ ggf. müssen Signale gesperrt werden

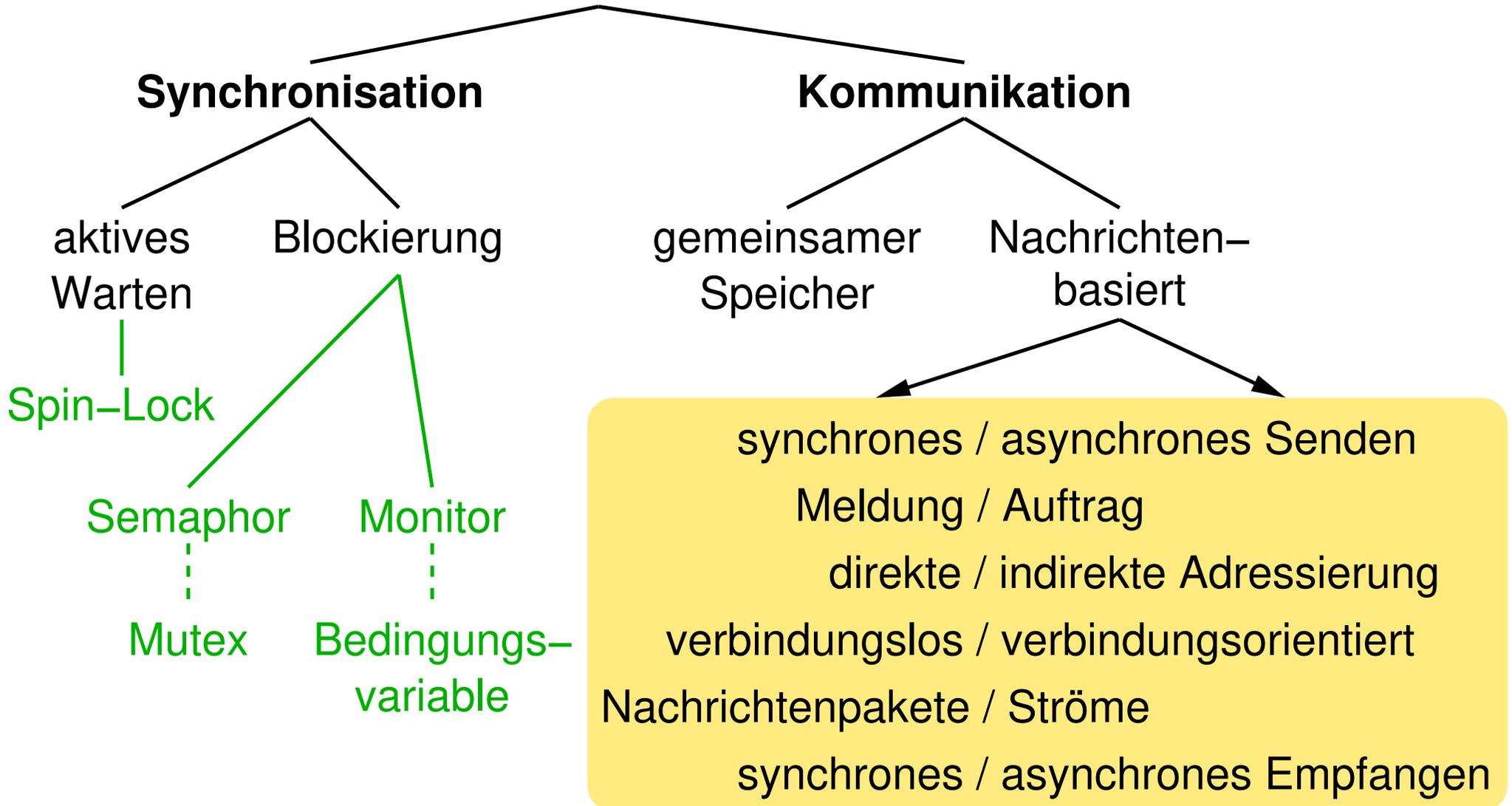


### Signale in POSIX

- ➔ Senden eines Signals an einen Prozeß (nicht Thread!):
  - ➔ `kill(pid_t pid, int signo)`
- ➔ Für jeden Signal-Typ ist eine Default-Aktion definiert, z.B.:
  - ➔ SIGINT: Terminieren des Prozesses (^C)
  - ➔ SIGKILL: Terminieren des Prozesses (nicht änderbar!)
  - ➔ SIGCHLD: Ignorieren (Kind-Prozeß wurde beendet)
  - ➔ SIGSTOP: Stoppen (Suspendieren) des Prozesses (^Z)
  - ➔ SIGCONT: Weiterführen des Prozesses (falls gestoppt)
- ➔ Prozeß kann Default-Aktionen ändern und eigene Signal-Handler definieren
- ➔ Handler wird von beliebigem Thread des Prozesses ausgeführt



## Threadinteraktion





- ➔ Nachrichtenbasierte Kommunikation
  - ➔ Primitive `send` und `receive`
  - ➔ synchrones / asynchrones Senden, Meldung / Auftrag
  - ➔ Mailboxen und Ports: indirekte Adressierung
  - ➔ Kanäle: logische Verbindung zwischen zwei Ports
    - ➔ Spezialfall: Ströme zur Übertragung von Sequenzen von Zeichen (z.B. POSIX Pipes)
  - ➔ RPC:
    - ➔ synchroner Auftrag, syntaktisch wie Prozeduraufruf
    - ➔ generierte Client- und Server-Stubs für *Marshaling*
  - ➔ Signale
    - ➔ asynchrone Benachrichtigung eines Prozesses
    - ➔ Unterbrechung eines Threads  $\Rightarrow$  führt Signal-Handler aus