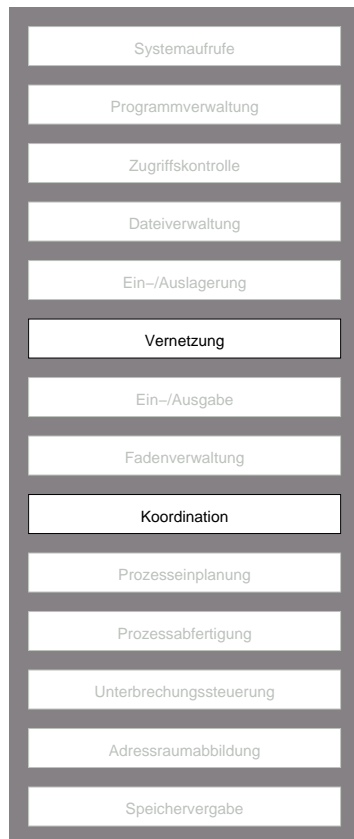


# Interprozesskommunikation



*inter-process communication* (IPC) Transfer von Daten von einem Prozess zu einem anderen Prozess . . .

- über gemeinsame Variablen (X Kap. 7)
  - die Prozesse haben einen gemeinsamen Datenbereich
  - „*shared memory* Kommunikation“
- über einen gemeinsamen Kommunikationskanal
  - der gemeinsame Datenbereich ist keine Voraussetzung
  - stattdessen werden Nachrichten versandt/empfangen
- mit impliziten Synchronisationseigenschaften

## Nachrichtenaustausch — *Message Passing*

- an die Stelle des Schreibens/Lesens von Datenwerten bei der Kommunikation über gemeinsame Variablen tritt das . . .

**Senden**      (*send*)  
**Empfangen**   (*receive*)   } von Nachrichten (auch Botschaften)

- zwischen beiden Aktionen besteht implizit eine *Sequentialitätsbeziehung*  
    ☞ eine Nachricht kann erst empfangen werden, nachdem sie gesendet wurde
- im Zeitverlauf muss der Sende- dem Empfangsbefehl aber nicht vorangehen:  
    zum Austausch der Daten sind Sender und Empfänger zu synchronisieren

# Prinzipielle Aktionen

**Datentransfer** vom Sende- zum Empfangsprozessadressraum

- *Botschaftenaustausch* über einen gemeinsamen Kommunikationskanal

**Synchronisation** von Sende- und Empfangsprozess

- der Fortschritt des Empfangsprozesses hängt ab vom Sendeprozess
  - Nachrichten sind konsumierbare Betriebsmittel
  - Sendeprozesse sind *Produzenten*, Empfangsprozesse sind *Konsumenten*
- der Fortschritt des Sendeprozesses hängt ab vom Empfangsprozess
  - Nachrichten werden über wiederverwendbare Betriebsmittel transportiert
  - typische „Transportbetriebsmittel“ sind Puffer (*bounded buffer*, X Kap. 7)
- die Koordination geschieht implizit mit der angewandten Primitive

# Grundlegende Operationen

**send** stellt eine Nachricht zum Empfang bereit

- blockierend (bis *receive/reply*) oder nicht-blockierend für den Sendeprozess

**receive** nimmt eine zum Empfang bereitgestellte Nachricht entgegen

- meist blockierend für den Empfangsprozess, aber auch nicht-blockierend

**reply** beantwortet eine entgegengenommene *Anforderungsnachricht* [38]

- nicht-blockierend für den Sendeprozess (der die Nachricht entgegen nahm)

**relay** stellt eine entgegengenommene *Anforderung* erneut zum Empfang bereit

- nicht-blockierend für den weiterleitenden Prozess [Warum blockieren *reply* und *relay* nicht?]

# Selektiver Nachrichtenaustausch

**selektives Empfangen** aus der Menge (vom System) empfangener Nachrichten wird diejenige vom Prozess angenommen, die bestimmte Kriterien erfüllt

- z. B. Dringlichkeit, Inhalt, Länge, Verweildauer, Alter, Absender, . . .
- bei mehreren akzeptablen Nachrichten wird die jüngste/älteste ausgewählt
- der Empfangsprozess wartet ggf. solange, bis die passende Nachricht eintrifft

**selektives Senden** eine zum Versand vorgesehene Nachricht wird erst bei Vorliegen eines Empfangsangebots verschickt

- bei mehrere Angeboten wird selektiert oder auch allen entsprochen
- evtl. wird die Nachricht (im System) zwischengepuffert und später gesendet
- der Sendeprozess wartet ggf. solange, bis ein passendes Angebot vorliegt

# Message Passing

IPC  $\iff$  E/A

**prozessororientierte Betriebssysteme** verwenden IPC als zentrales Paradigma

- typische Betriebssystemdienste sind als *Systemprozesse* ausgelegt
  - SoS<sub>i</sub>-System (X Kap. 5): Ebenen 1, 2, 7–14
- ein *Mikrokern* implementiert die grundlegenden Primitiven zur IPC
  - SoS<sub>i</sub>-System (X Kap. 5): Ebenen 3–6

☞ Thoth, . . . , V, Chorus, Mach, Amoeba, . . . , Peace, L4, DROPS

**prozedurorientierte Betriebssysteme** bieten IPC lediglich als Dienstleistung

- die Primitiven sind nicht optimiert für prozessororientierte Systeme
- sie entsprechen vielmehr spezialisierten E/A-Funktionen ☞ UNIX & Co

`s = socket (domain, type, protocol)` ein Kommunikationsanschluss

- liefert den „Sockel“ *s* zum Aufbau einer *Kommunikationsverbindung*
- selektiert die Domäne (z. B. UNIX, Internet) und ein konkretes Protokoll
- assoziiert die Kommunikationssemantik (z. B. Strom, Datagramm)

`ok = bind (s, name, namelen)` benennt einen Kommunikationsanschluss

- macht den (lokalen) Sockel *s* über einen Namen global bekannt

`ok = close (s)` streicht einen Kommunikationsanschluss

`ok = connect (s, name, namelen)` initiiert eine Verbindung, je nach s

- `name{,len}` identifiziert  $\left\{ \begin{array}{ll} \text{die Empfangsseite} & \text{☞ Datagramm } (\neg \text{Verbindung}) \\ \text{das Verbindungsende} & \text{☞ Strom} \end{array} \right.$

`ok = listen (s, backlog)` eröffnet Bereitschaft zur Verbindungsannahme

- `backlog` definiert die max. Anzahl anhängender Verbindungsaufbauwünsche

`as = accept (s, addr, addrlen)` nimmt eine Verbindung ( $\{, \neg\}$ block.) an

- liefert einen neuen Sockel `as` mit den gleichen Eigenschaften wie `s`
- `addr{,len}` enthält als Ergebnis die Adresse der verbindenden Instanz

`ns = send (s, msg, len, flags)` sendet eine Nachricht

- Sockel `s` muss mit einem anderen Sockel verbunden sein
- je nach Sockeltyp blockiert die Funktion bei zu knappem Pufferplatz
- `flags` spezifiziert ggf. die Sendesemantik (*out of band, bypass routing*)
- Ergebnis ist die Anzahl der gesendeten Bytes (`ns`)

`nr = recv (s, msg, len, flags)` empfängt eine Nachricht

- analog zu `send()` ...

☞ `read(2), write(2), sendto(2), recvfrom(2), sendmsg(2), recvmsg(2)`

# Kommunikationsmodelle

**gleichberechtigte Kommunikation** die Prozesse spielen *dieselbe Rolle*; beide Kommunikationspartner sind sowohl Sender als auch Empfänger

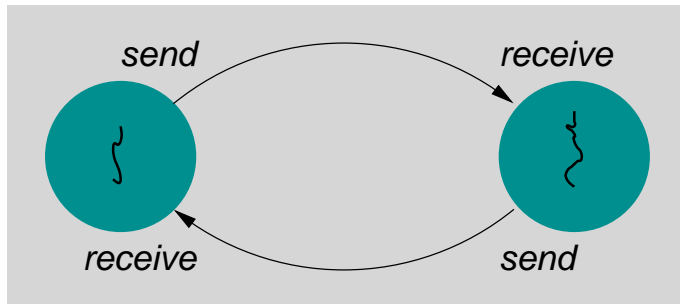
$$P_1 \left\{ \begin{array}{ccc} \textit{send} & \longrightarrow & \textit{receive} \\ \textit{receive} & \longleftarrow & \textit{send} \end{array} \right\} P_2$$

**ungleichberechtigte Kommunikation** die Prozesse spielen *verschiedene Rollen*; ein Kommunikationspartner ist **Auftraggeber**, der andere ist **Auftragnehmer**

$$P_1 \left\{ \begin{array}{ccc} \textit{send} & \longrightarrow & \textit{receive} \\ & \longleftarrow & \textit{reply} \end{array} \right\} P_2$$

☞ *Client/Server-Modell*

# Gleichberechtigte Kommunikation



Die an der Kommunikation beteiligten Prozesse sind in ihrer Rollenfunktion gleichzeitig Produzent und Konsument. **Blockierendes Senden** bedeutet höchste *Verklemmungsgefahr*. [Warum?]

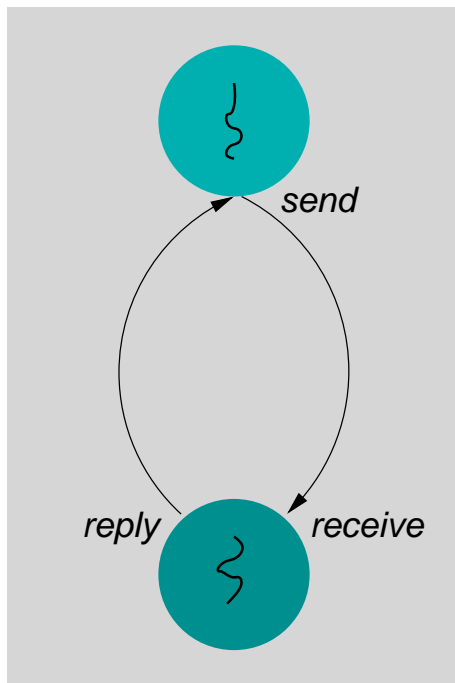
**Annahme:** *send* verfügt nicht über die erforderlichen Betriebsmittel (☞ *bounded buffer*). Als Folge davon könnten sich die Prozesse aufeinander und anhaltend blockieren (☞ *deadlock*).

```
void thread () {  
    :  
    send("fibril", data);  
    :  
}
```

```
void fibril () {  
    :  
    send("thread", data);  
    :  
}
```

☞ nicht-blockierendes Senden bzw. ein „ausgeklügeltes“ *Protokoll* fahren

# Ungleichberechtigte Kommunikation



Die an der Kommunikation beteiligten Prozesse besitzen unterschiedliche Rollenfunktionen:

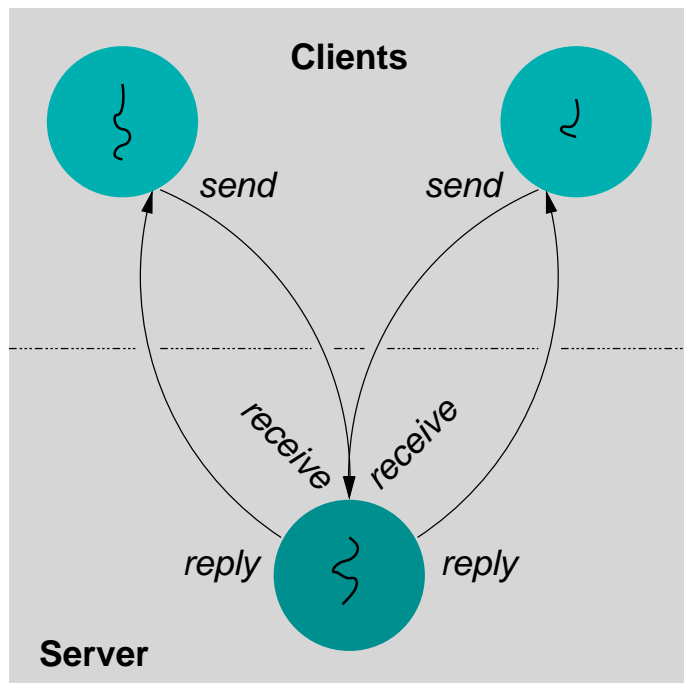
**Auftraggeber** fordert mit *send* die Durchführung einer Aufgabe an und erwartet eine Antwort

**Auftragnehmer** nimmt mit *receive* eine Aufforderung zur Aufgabenbearbeitung entgegen, führt die Aufgabe aus und sendet mit *reply* eine Antwort zurück

☞ *send* und *receive* blockieren, *reply* nicht

[Warum?]

# Client/Server-Kommunikation (1)



**Client**  $\equiv$  Auftraggeber

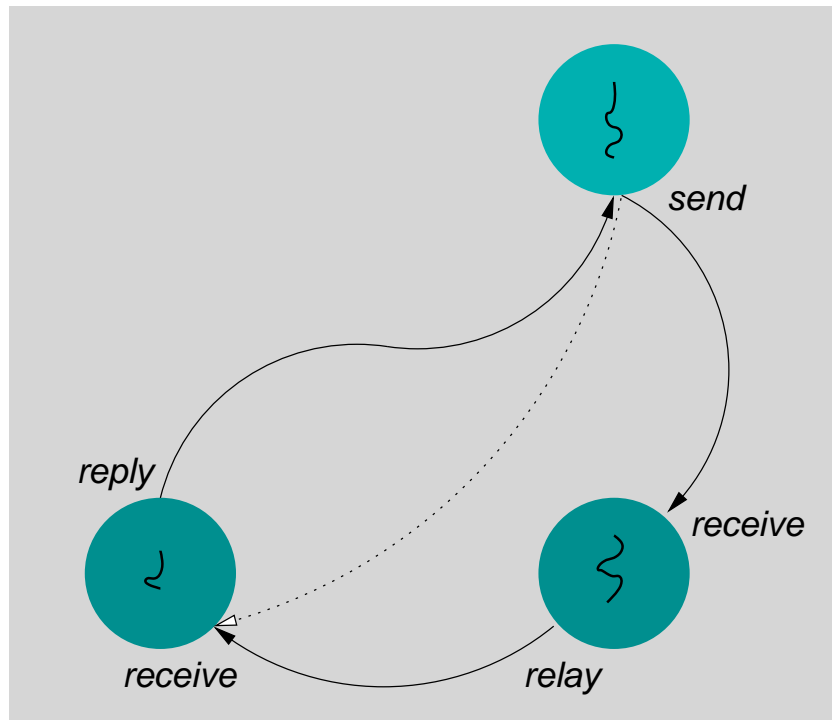
- Rolle „Dienstnutzer“ (*service user*)
- 1 Dienst wird von 1 bzw.  $N$  Klienten genutzt

**Server**  $\equiv$  Auftragnehmer

- Rolle „Dienstleister“ (*service provider*)
- 1 Dienst wird von 1 bzw.  $N$  Server betrieben

☞  $\{ \textit{file}, \textit{mail}, \textit{print}, \textit{time}, \textit{web}, \dots \}$  *service*

## Weiterleitung von Anforderungsnachrichten



Ein Prozess gibt sein „Antwortrecht“ auf ein *send* (☞ *reply*) mit *relay* ab:

- das Recht wird durch *receive* erworben
- die weitergeleitete Nachricht muss neu empfangen werden (ggf. selbst von dem weiterleitenden Prozess)
- der im *send* blockierte Prozess erwartet die Antwort von dem die Nachricht neu empfangenden Prozess

☞ Weiterleitung verändert ggf. die ursprünglich mit *send* verschickte Nachricht

## *reply* $\neq$ *send*

```
int rr_send (int peer, char *data, unsigned size) {  
    send(peer, data, size);  
    return receive(data, size);  
}
```

```
int rr_receive (char *data, unsigned size) {  
    return receive(data, size);  
}
```

```
int rr_reply (int peer, char *data, unsigned size) {  
    return send(peer, data, size);  
}
```

Auftragnehmer müssen immer verfügbar sein, was im Beispiel aber nicht gewährleistet ist:

- das „nachgebildete *reply*“ blockiert den Auftragnehmer, wenn das Betriebsmittel zur Antwortaufnahme fehlt (☞ *bounded buffer*)
- das „native *reply*“ blockiert den Auftragnehmer nicht, da hier das Betriebsmittel zur Antwortaufnahme verfügbar ist [Welches?]

[Warum ist das alles ein Problem?]

# Prozeduren als Kommunikationsmittel

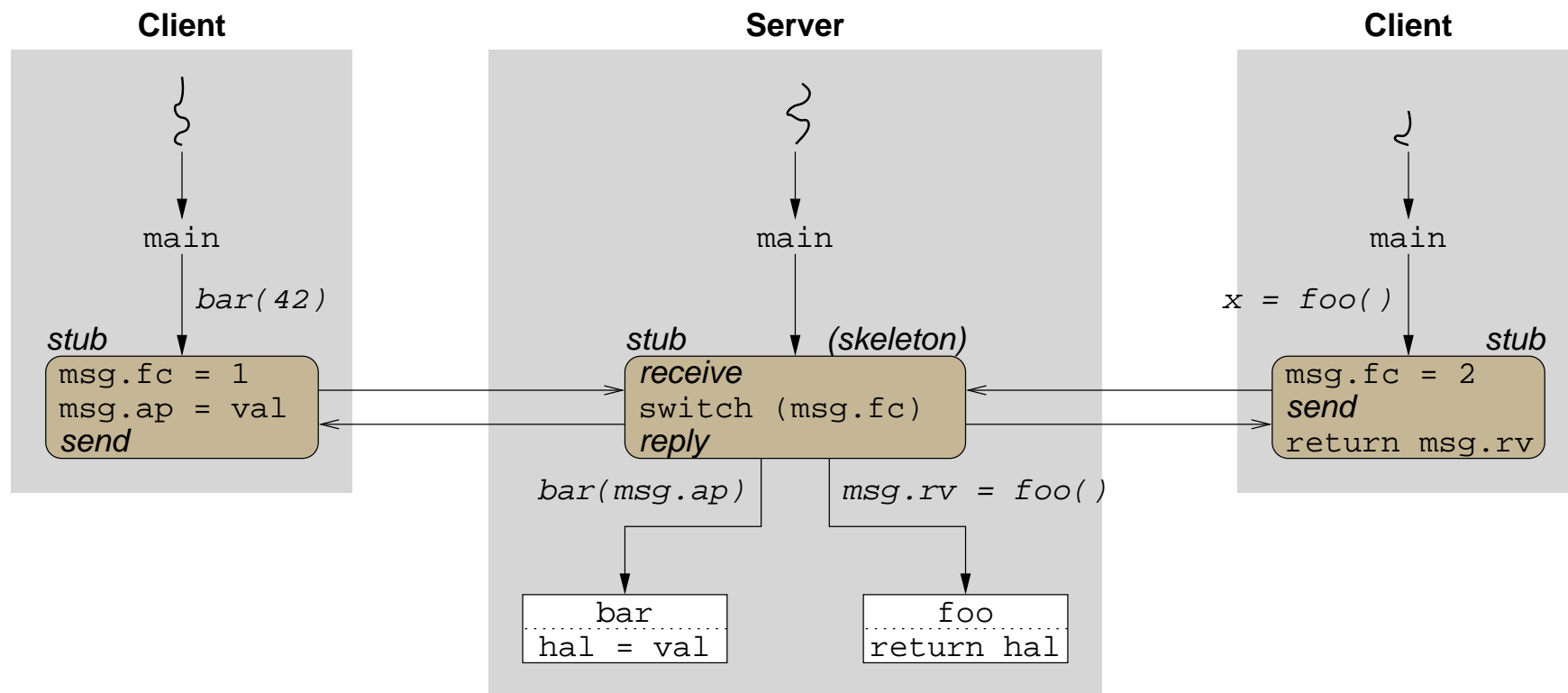
**Prozedurfernaufruf** (*remote procedure call*, RPC [9]) — Ein Prozess (☞ Client) ruft eine von einem anderen Prozess (☞ Server) bereitgestellte Prozedur auf, die in einem anderen Datenbereich (☞ Adressraum) arbeitet

- syntaktisch besteht kein Unterschied zum „normalen“ Prozeduraufruf *Ideal*
  - für jede ferne Prozedur existiert beim Client ein „Stumpf“ (*client stub*)
  - so auch beim Server für alle fernen Aufrufstellen (*server stub, skeleton*)
  - in den Stümpfen läuft beim Aufruf die Client/Server-Kommunikation ab

☞ ferne Prozeduren  $\left\{ \begin{array}{l} \text{werden lokal aufgerufen} \\ \text{kehren an ihre Aufrufstellen lokal zurück} \end{array} \right.$

- der *semantische Unterschied* ist groß: Parameterübergabe, Fehlermodell

## Client/Server-Kommunikation (2)



# Rendezvouskommunikation

**Ren·dez·vous 1** Treffen (von Verliebten) **2** Begegnung (von Satelliten); zwei Prozesse treffen sich „kurzzeitig“ zur (uni-/bidirektionalen) Datenübergabe und gehen danach wieder getrennte Wege

- der eine Prozess beantragt ein Rendezvous (*request*)
- der andere Prozess . . .
  - bietet ein oder mehrere Rendezvous an (*offer*),
  - nimmt eins der beantragten Rendezvous entgegen (*acceptance*) und
  - führt das angenommene Rendezvous aus (*execution*)
- strikt synchrones, bevorzugt sprachgestütztes Paradigma (☞ Ada)
  - Werte von Programmvariablen werden auf direktem Wege ausgetauscht
  - der Datentransfer verläuft „End-zu-End“, ohne Zwischenpufferung

## *copy on write* — COW

- Kommunikation beeinflusst die „Fähigkeit“ (*capability*) eines Prozesses:  
*send* entzieht dem aktuellen Prozess (Sender) das Schreibzugriffsrecht auf die in seinem Adressraum vorliegende Nachricht  
*receive* übergibt dem aktuellen Prozess (Empfänger) das Lesezugriffsrecht auf die im Senderadressraum vorliegende Nachricht
- Kopieren wird zum Ausnahmefall [39]: *segment swapping* bzw. *paging*  
– wenn der  $\left\{ \begin{array}{c} \text{Sender} \\ \text{Empfänger} \end{array} \right\}$  nach dem  $\left\{ \begin{array}{c} \textit{send} \\ \textit{receive} \end{array} \right\}$  die Nachricht modifiziert
- Nachrichten müssen vom Betriebssystem als **Segmente** verwaltet werden

# Kommunikationsverläufe

## **synchron** und blockierend

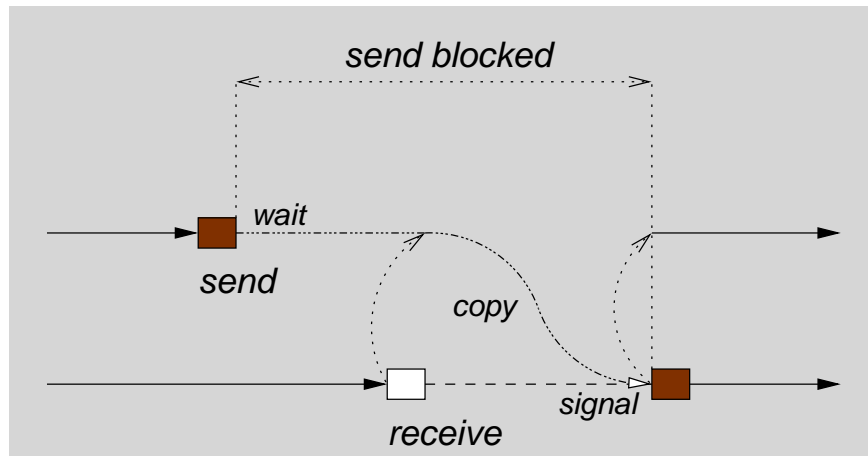
- Sendeprozess fordert im *send* das Betriebsmittel „Empfangsprozess“ an
- Empfangsprozess fordert im *receive* das Betriebsmittel „Sendeprozess“ an
- Unterstützung von „End-zu-End“ Datentransfers ohne Zwischenpufferung

## **asynchron** und blockierend oder nicht-blockierend

- Sendeprozess fordert im *send* das Betriebsmittel „Puffer“ an
- Empfangsprozess fordert im *receive* das Betriebsmittel „Nachricht“ an
- Unterstützung bzw. Ausnutzung von Fließbandverfahren (*pipelining*)

## **zuverlässig** garantierter Datentransfer von Sende- zu Empfangsprozess

## Synchrone IPC



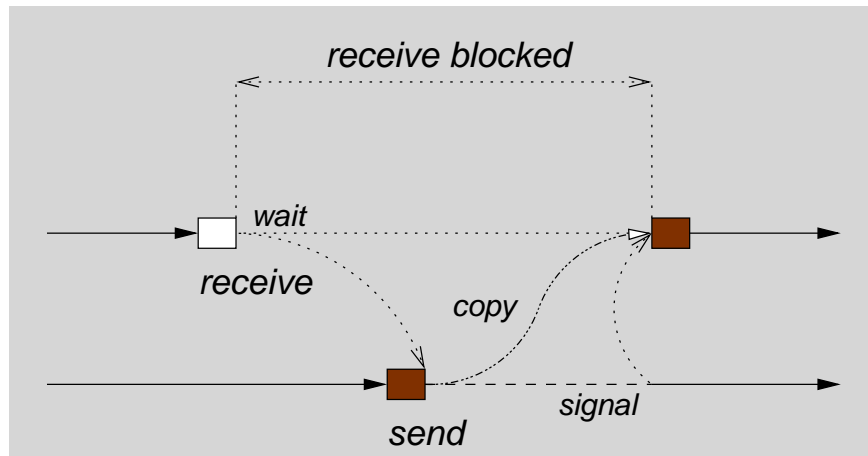
## Sendeseitige Synchronisation

Der Sendeprozess wartet im *send* auf das *receive* des Empfangsprozesses, d. h., bis das zur Nachrichtenaufnahme benötigte Betriebsmittel (☞ Empfangspuffer) zur Verfügung steht und der Datentransfer vollzogen ist.

- aktive und kontrollierende Instanz ist der Empfangsprozess
  - er initiiert den Datentransfer und signalisiert den Sendeprozess
- *einseitige Synchronisation* des Sendeprozesses mit dem Empfangsprozess

## Synchrone IPC

## Empfangsseitige Synchronisation

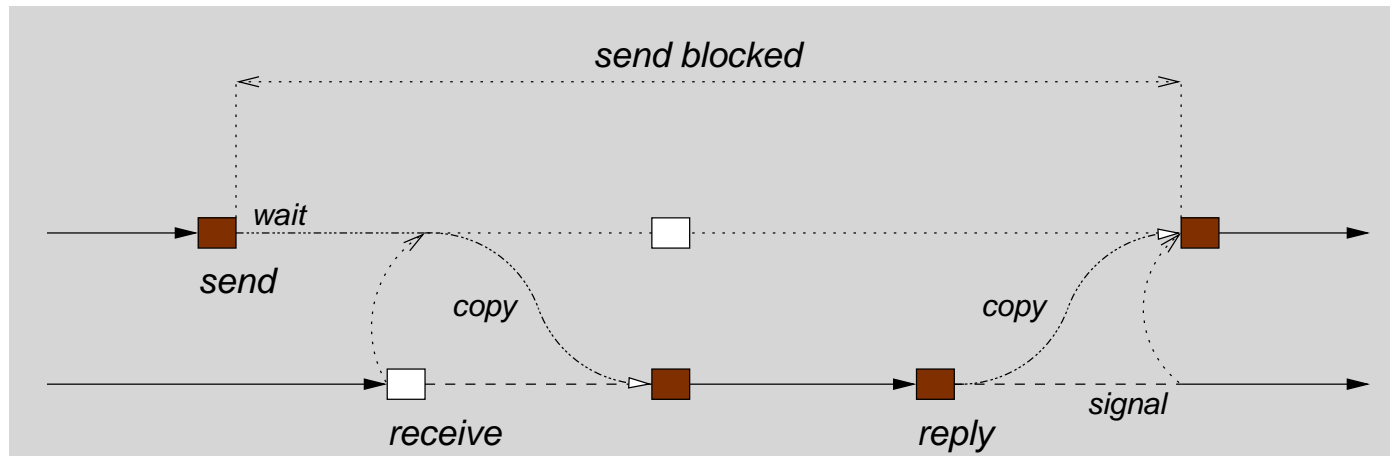


Der Empfangsprozess wartet im *receive* auf das *send* des Sendeprozesses, d. h., bis das für sein weiteres Vorkommen benötigte Betriebsmittel (☞ Nachricht) zur Verfügung steht und der Transfer der Daten vollzogen ist.

- aktive und kontrollierende Instanz ist der Sendeprozess
  - er initiiert den Datentransfer und signalisiert den Empfangsprozess
- *einseitige Synchronisation* des Empfangsprozesses mit dem Sendeprozess

## Synchrone IPC

## Client-seitige Synchronisation

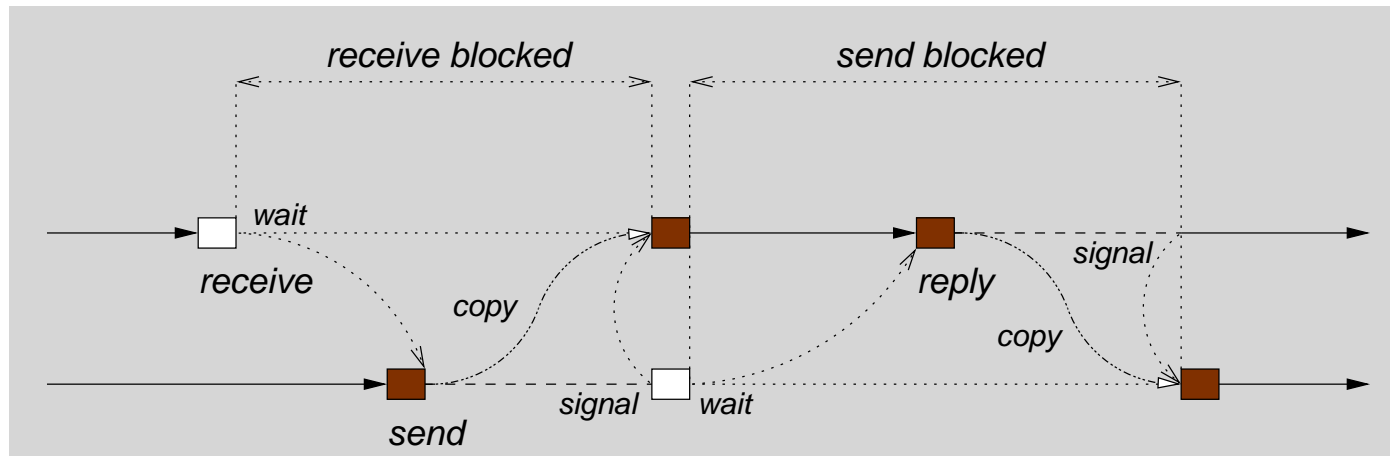


Der Client wartet im *send* auf das *reply* des Servers, der die Nachricht beantworten wird.

- interprozedurale Beziehungen können Prozess übergreifend ausgelegt werden
  - *send* „kodiert“ den Prozeduraufruf, *reply* den Prozedurrücksprung
- „*request-reply* Ansatz“: der Client fordert dem Server eine Antwort ab

## Synchrone IPC

## Server-seitige Synchronisation

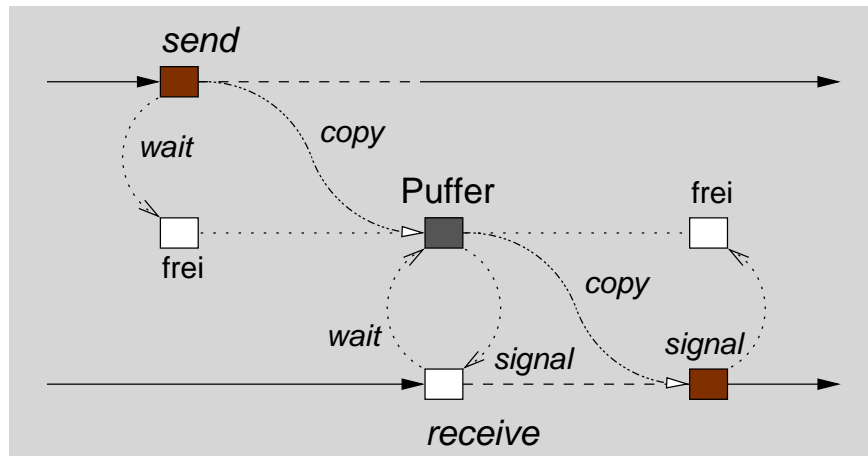



Der Server wartet im *receive* auf das *send* eines Clients, um eine Nachricht zu bearbeiten und ggf. mit *reply* zu beantworten.

- Nachricht empfangender und antwortender Prozess können verschieden sein
  - wenn die Empfangsnachricht mit *relay* einem anderen Server zugestellt wird
- ein Server kann selbst Client eines anderen Servers sein

## Asynchrone IPC

## Pufferblockierendes Senden

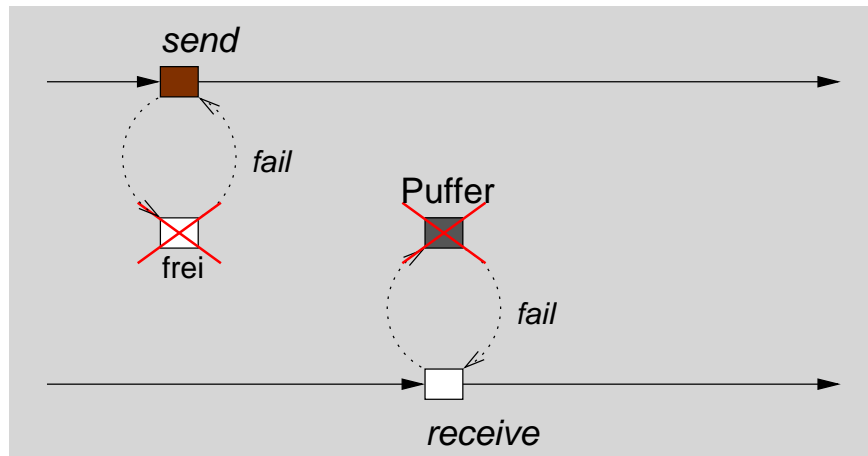


Sende- und Empfangsprozess benutzen einen Puffer ( *bounded buffer*) zum Datentransfer. In Ausnahmefällen müssen sie jedoch warten: der Sendeprozess auf ein wiederverwendbares Betriebsmittel, der Empfangsprozess dagegen auf ein konsumierbares Betriebsmittel.

- aktive und kontrollierende Instanz ist Sende- und Empfangsprozess
  - beide initiieren Kopiervorgänge und signalisieren den jeweils anderen
- (potentiell) *blockierende Semantik* garantiert ein Gelingen des Datentransfers

## Asynchrone IPC

## Unzuverlässiges Senden

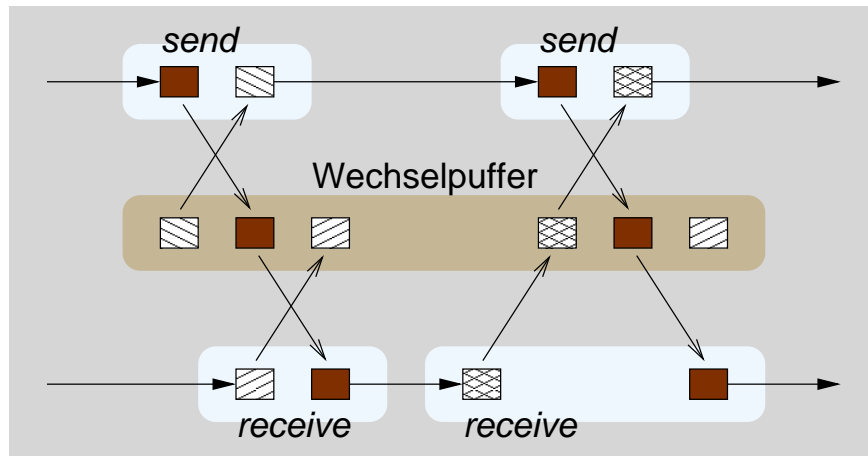


Sende- und Empfangsprozess benutzen einen gemeinsamen Puffer zum Transfer der Daten. In Ausnahmefällen scheitert jedoch die Operation: das *send*, wenn kein Puffer mehr frei ist und das *receive*, wenn keine Nachricht zwischengepuffert ist.

- beide Prozesse müssen die Operationen wiederholen, um erfolgreich zu sein
  - ggf. warten sie aktiv auf die Betriebsmittelbereitstellung durch den anderen
- *nicht-blockierende Semantik* garantiert kein Gelingen des Datentransfers

## Asynchrone IPC

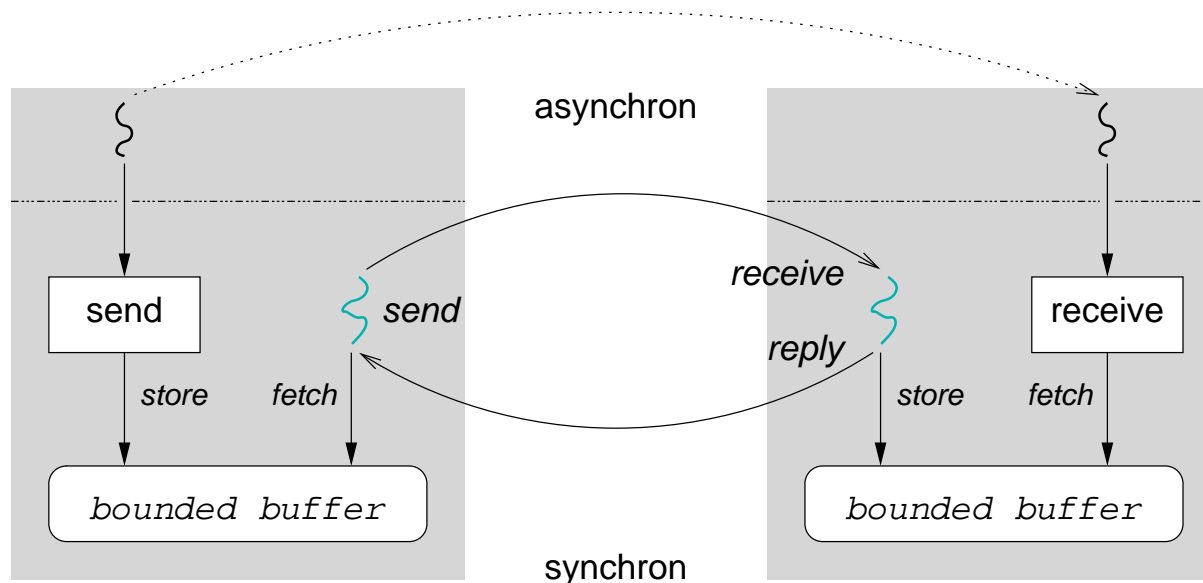
## Nicht-kopierender Datentransfer



Der Sendeprozess bietet einen gefüllten Puffer an und erhält einen leeren zurück. Umgekehrt bietet der Empfangsprozess einen leeren Puffer an und erhält einen gefüllten zurück. Daten werden so ohne Kopiervorgänge von Sender zu Empfänger transferiert.

- verfügt das „System“ über keinen *Wechselpuffer*, liegt ein Ausnahmefall vor
  - die Operationen blockieren (☞ zuverlässig) oder scheitern (☞ unzuverlässig)
- ist performant, wenn Adressraumgrenzen nicht überschritten werden müssen

## Synchron vs. Asynchron



## Dualität von IPC

Asynchrone IPC ist durch „Hilfsfäden“ abbildbar auf synchrone IPC. Die Hilfsfäden kontrollieren eine *Pipeline*, die die Hauptfäden voneinander entkoppelt.

- der Ansatz kombiniert die Vorteile von synchroner IPC und asynchroner IPC
  - effizienter Nachrichtenaustausch und höherer Grad an Nebenläufigkeit
- der umgekehrte Fall funktioniert auch, wirft jedoch Probleme auf (X 8-15)

[Welche?]

# Kommunikation und Betriebsmittel

- Prozesse synchronisieren sich zur Bereitstellung von Betriebsmitteln

**Sender** benötigt das wiederverwendbare Betriebsmittel „Puffer“

synchrone IPC  $\Rightarrow$  in den Zielpuffer

asynchrone IPC  $\Rightarrow$  in den Zwischenpuffer

**Empfänger** benötigt das konsumierbare Betriebsmittel „Nachricht“

asynchrone IPC  $\Rightarrow$  aus dem Zwischenpuffer

synchrone IPC  $\Rightarrow$  aus dem Quellpuffer

- ohne Betriebsmittel „Puffer“ und „Nachricht“ scheitert die Kommunikation

# Adressierung

**Port** (*port*) ein *Anschluss* zur Weiterleitung/Zustellung von Nachrichten, der einem bestimmten Prozess zugeordnet ist

- die Zuordnung ist statisch (☞ Prozesserzeugung) oder dynamisch
- Prozesse können mehrere Anschlüsse besitzen: Ein- und/oder Ausgänge
- realisiert eine „enge Kopplung“ zwischen kooperierenden Prozessen

**Briefkasten** (*mailbox*) ein *Zwischenspeicher* für Nachrichten, der durch *send* gefüllt und durch *receive* geleert wird

- der Pufferbereich (☞ *bounded buffer*) ist keinem Prozess zugeordnet
- viele Prozesse können daraus lesen (*receive*) und dahin schreiben (*send*)
- realisiert eine „lose Kopplung“ zwischen kooperierenden Prozessen

[In Welche Kategorie fällt *socket* (2)?]

# Verbindungen

Eine Alternative, bei der (mindestens) zwei Ports miteinander verknüpft werden müssen, um IPC zu ermöglichen. Drei Phasen werden dabei unterschieden:

**Aufbauphase** reserviert Betriebsmittel (Puffer, Fäden), erstellt die Verbindung je nach Portfunktion:

<i>unidirektional</i>	Port <sub>send</sub>	→	Port <sub>receive</sub>
<i>bidirektional</i>	Port <sub>send/receive</sub>	↔	Port <sub>receive/send</sub>

**Nutzungsphase** bedeutet IPC je nach Bedarf und Verbindungseigenschaften

**Abbauphase** gibt die Betriebsmittel frei, löst die Verbindung auf

# Zusammenfassung

- Kommunikationsprimitiven: *send*, *receive*, *reply*, *relay*, . . . , UNIX & Co
- Kommunikationsmodelle:
  - (un)gleichberechtigte bzw. Client/Server-Kommunikation
  - Prozedurfernaufruf, Rendezvous, *copy on write*

- Kommunikationsverläufe:

nicht-blockierend  $\rightarrow$   $\left. \begin{array}{l} \text{synchron} \\ \text{asynchron} \end{array} \right\} \rightarrow \text{blockierend} \rightarrow \left\{ \begin{array}{l} \text{gepuffert} \\ \text{ungepuffert} \end{array} \right.$

- Kommunikationskanäle: Port, Briefkasten, Portverbindung