Vorlesung Betriebssysteme I

Thema 6: Kommunikation

Robert Baumgartl

14. Dezember 2015

Einige Gedanken

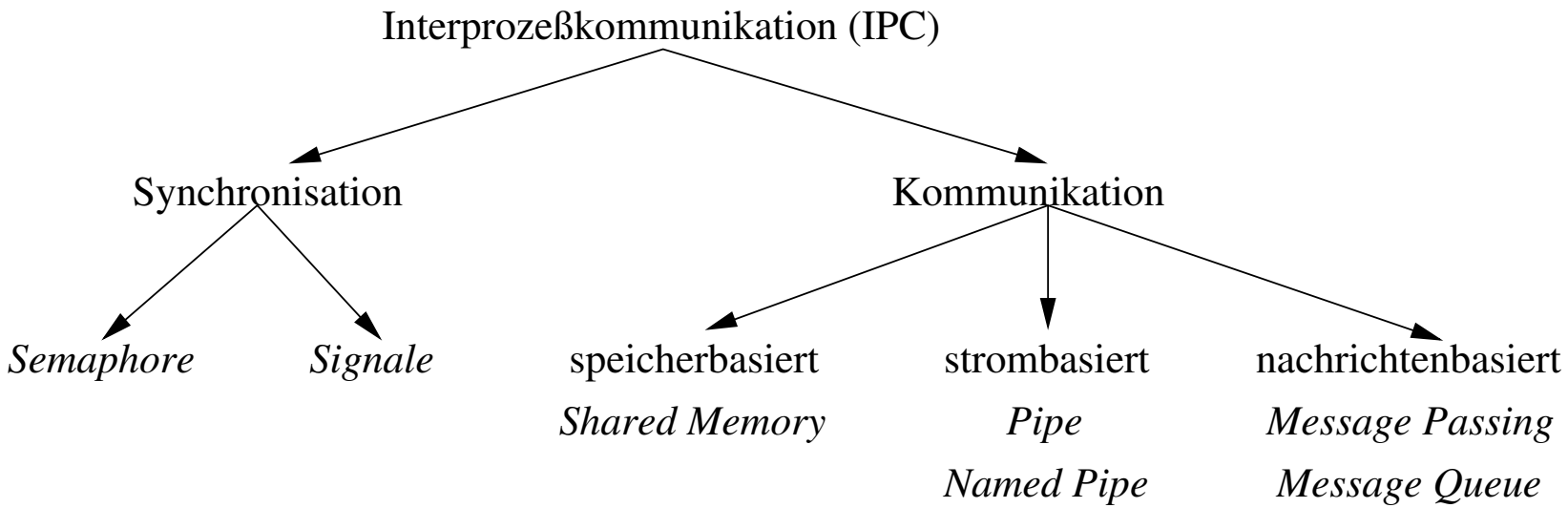
Kommunikation = Übertragung von Informationen zwischen Aktivitäten

- meist mit Synchronisation (d. h., zeitlicher Koordination) verbunden
- Synonym: *Inter Process Communication* (IPC)
- Vielzahl an Mechanismen, „historisch gewachsen“
- Teilnehmer benötigen gemeinsam genutzte Ressource
- Identifikation/Authentisierung der Teilnehmer erforderlich

Beispiele für IPC-Mechanismen (Auswahl)

- Datei
- Pipe
- Signal
- benannte Pipe (FIFO)
- Socket
- gemeinsam genutzer Speicher (*Shared Memory*)
- Nachrichten
- Mailboxen
- speichereingeblendete Datei (*memory-mapped File*)
- entfernter Prozeduraufruf (*Remote Procedure Call*)
- Clipboard
- ...

Kategorisierung von IPC-Mechanismen



Kommunikationsbeziehungen

Anzahl der Teilnehmer:

- 1:1
- 1:m
- n:1
- n:m

Gleichzeitigkeit von Hin- und Rückkanal:

- unidirektional
- bidirektional

weitere Aspekte:

- lokale vs. entfernte Kommunikation
- direkte vs. indirekte Kommunikation

Synchrone und Asynchrone Kommunikation

Sendeoperation (*Send*)

- synchron: Sender wartet (blockiert), bis Empfänger die Information entgegengenommen hat (implizite Quittung)
- asynchron: Sender setzt nach dem Senden einer Nachricht sofort seine Arbeit fort („No-Wait-Send“, „Fire and Forget“); Beispiel: Telegramm

Empfangsoperation (*Receive*)

- synchron: Die Empfangsoperation blockiert den Empfänger so lange, bis Information eintrifft.
- asynchron: Empfänger liest Information, falls empfangen wurde und arbeitet anschließend weiter, auch wenn nichts empfangen wurde

Synchrone vs. Asynchrone Operationen

synchrone Operationen:

- direkte Zustellung der Informationen (ohne Zwischenspeicher)
- implizite Empfangsbestätigung
- i. a. einfacher zu programmieren
- standardmäßig arbeiten Kommunikationsoperationen in Unix synchron

asynchron:

- Nachricht muss zwischengespeichert werden
- Vorteil: wenn ein kommunizierender Prozess abbricht, dann wird der Partner nicht unendlich blockiert
- kein *Deadlock* möglich (gegenseitige Blockierung infolge fehlerhafter Programmierung)
- Benachrichtigung des Empfängers u. U. kompliziert

Verbindungsorientierte und verbindungslose Kommunikation

	verbindungsorientiert	*verbindungslos*
Ablauf	3 Phasen: Aufbau der Verbindung Datenübertragung Abbau der Verbindung	1 Phase: Datenübertragung
Beispiele	(analoges) Telefon TCP Pipe	Telegramm IP Signal

Verbindungsarten

Unicast	Punkt-zu-Punkt-Verbindung, Direktverbindung, 2 Teilnehmer
Multicast	1 Sender, mehrere (aber nicht alle) Empfänger, Gruppenruf
Broadcast	1 Sender, alle Empfänger (z. B. eines Subnetzes)

Kommunikation über Datei

- ältester IPC-Mechanismus
- Sender schreibt Daten in Datei
- Empfänger liest Daten aus Datei
- nachteilig: zweimaliger Zugriff auf Massenspeicher
- aber: es gibt auch Dateisysteme im RAM
- nachteilig: überlappender Zugriff durch Sender und Empfänger
- Lösung: Sperren der Datei (*File Locking*), z. B. mittels `lockf()`
- Beispiel: `filelock.c` (extern)
- Problem: Sperren setzt Wohlverhalten voraus

Kommunikation mittels Pipe („Röhre“)

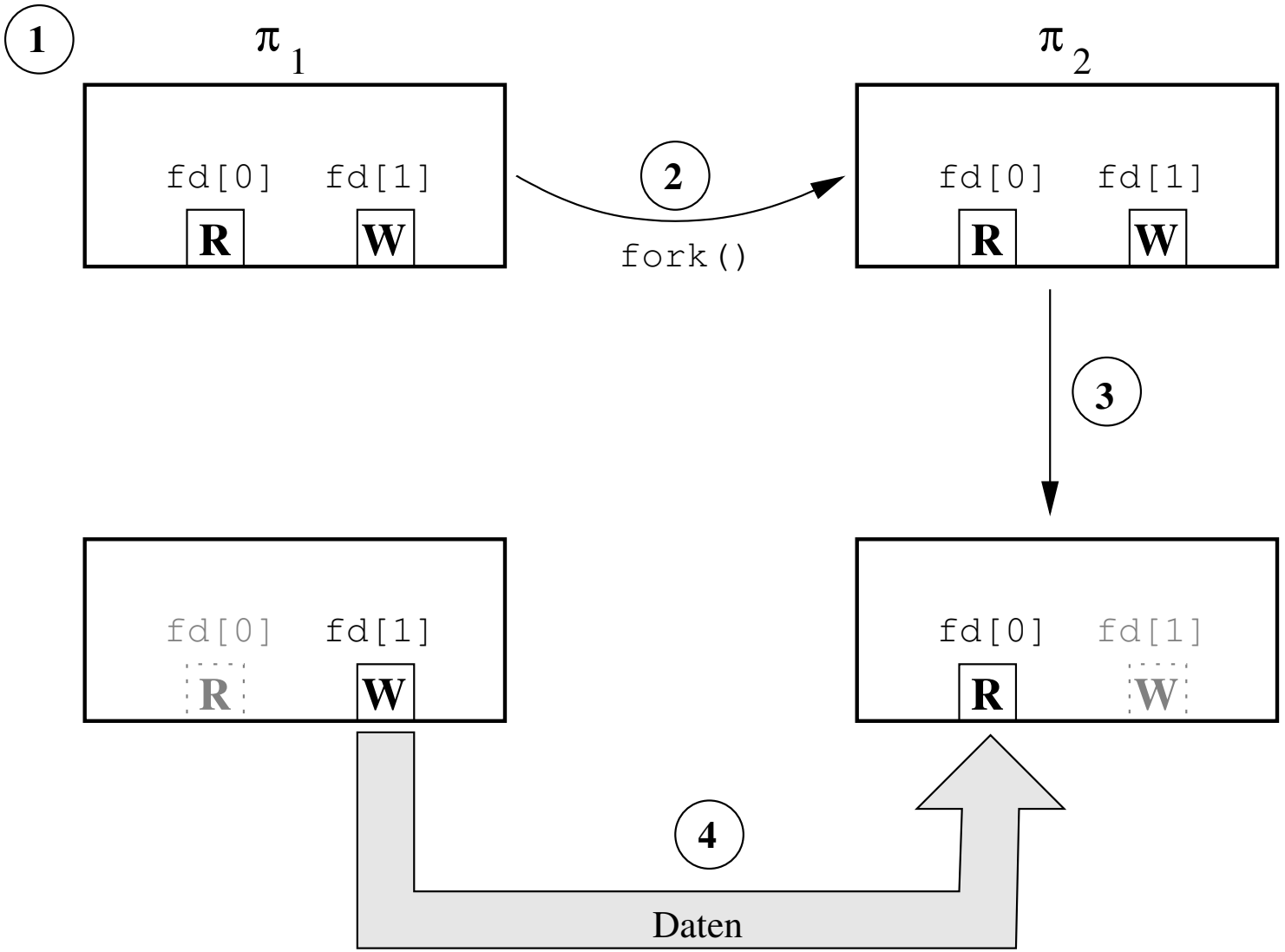
```
int pipe(int filedes[2]);
```

- liefert Feld von 2 Dateideskriptoren zurück (einer zum Lesen, einer zum Schreiben)
- Zugriff wie auf eine Datei (`read()`, `write()`, `close()`), jedoch kein `open()`, `lseek()`
- Datenübertragung innerhalb eines Prozesses sinnlos?!
- Woher weiß Empfänger, dass die Pipe existiert?

Vorgehensweise

1. Prozess ruft `pipe()` $\rightarrow$ Pipe wird durch BS angelegt.
2. Prozess ruft `fork()` (Deskriptoren werden vererbt!).
3. Jeder Prozess schließt einen seiner Deskriptoren. (Verständigung erfolgt)
4. Datenübertragung mittels `read()` bzw. `write()`
5. Beide Prozesse rufen `close()` $\rightarrow$ Pipe wird nach zweitem `close()` durch BS vernichtet

Veranschaulichung der Pipe-Erstellung



Eigenschaften der Pipe

- stets unidirektional ($\rightarrow$ für bidirektionale Übertragung 2 Pipes nötig)
- keine Synchronisation beim Schreiben (Daten werden im Kern gepuffert)
- Schreiboperationen, die weniger als `PIPE_BUF`[1] Daten umfassen, müssen *atomar* (in einem Rutsch) erfolgen.
- keine persistente Ressource (verschwindet nach letztem `close()`)
- nur zwischen verwandten Prozessen möglich!

[1] Linux gegenwärtig: 4096 Bytes; vgl. `limits.h`

Pipe: problematische Situationen

- Lesen von einer eröffneten Pipe, die keine Daten enthält, blockiert. Wird danach die Schreibseite geschlossen, kehrt `read()` mit Resultat 0 zurück.
- Leseoperation aus Pipe mit ungültigem Filedeskriptor – liefert Fehler EBADF (*Bad File Descriptor*)
- Leseoperation aus Pipe, die nach Schreibvorgang geschlossen wurde – liefert zunächst Daten, dann 0 als Resultat $\widehat{=}$ End of File (EOF)
- Schreiboperation auf Pipe, deren Leseseite geschlossen – liefert Signal `SIGPIPE` an schreibenden Prozess
- Prozess(e) enden mit eröffneter Pipe – Filedeskriptoren werden bei Prozessbeendigung automatisch geschlossen

Literatur: `man 7 pipe`

Anwendungsbeispiel

Pipe-Operator (‘|’) der Shell zur Verknüpfung von *stdout* des Senders mit *stdin* des Empfängers:

```
robge@ipaetz2:~$ du | sort -n -r | less
```

Beispiel 2: `simplepipe.c` (extern)

Putting it all together: `popen()`

```
FILE *popen(const char *cmd, const char *type);
```

- legt eine Pipe an, forkt den rufenden Prozess und ruft im Kind eine Shell auf, die `cmd` ausführt
- Resultat: Zeiger auf I/O-Strom, der
  - mit `stdin` von `cmd` verbunden ist, wenn `type == "w"` oder
  - mit `stdout` von `cmd` verbunden ist, wenn `type == "r"`.
  - Lese- oder Schreiboperation geschehen also mit Pipe, die mit ge-fork()-tem Prozess `cmd` verbunden ist
- muss mit `pclose()` geschlossen werden
- erleichtert Kommunikation C-Programm $\leftrightarrow$ Shellkommando

Beispiel: `popen.c` (extern)

Signale

- Mittel zur Signalisierung zwischen Prozessen bzw. BS und Prozessen
- Übermittlung einer Information, ohne dass Prozess *aktiv* beteiligt
- Ursprung: UNIX
- Generierung $\rightarrow$ Zustellung $\rightarrow$ Behandlung (Reaktion auf Signal)
- jedes Signal hat Nummer, Name, Defaultaktion
- meist (aber nicht immer) keine Datenübertragung
- Verwandschaft der Partner ist nicht notwendig

Signale – Prinzipieller Ablauf

1. Sende-Prozess generiert ein Signal
2. System stellt das Signal dem Empfänger-Prozess zu
3. Wenn Empfänger Signalhandler installiert hat $\rightarrow$ Aufruf des Signalhandlers (asynchron zur Programmausführung)
4. Wenn kein Signalhandler installiert $\rightarrow$ Ausführung der Default-Aktion (z. B. Abbruch, Ignorieren)

Signale unter Unix (Übersicht)

Name	*Def.-Aktion*	*Semantik*
SIGHUP	Abbruch	Verbindung beendet (Hangup)
SIGINT	Abbruch	CTRL-C von der Tastatur
SIGILL	Abbruch	Illegale Instruktion
SIGKILL	Abbruch	Sofortige Beendigung
SIGSEGV	Coredump	Segmentation Violation
SIGPIPE	Abbruch	Schreiben in ungeöffnete Pipe
SIGCHLD	Ignoriert	Kind-Prozess beendet
SIGSTOP	Stop	Anhalten des Prozesses
SIGTSTP	Stop	CTRL-Z von der Tastatur
SIGCONT		Fortsetzen eines angehaltenen Prozesses

Tabelle: Auswahl von Signalen nach POSIX

vollständige Übersicht: `man 7 signal`

Senden von Signalen an der Kommandozeile

Senden mit dem (externen) Kommando `kill`:

```
robge@hadrian:~$ while true ; do echo -n; done &
[1] 6578
robge@hadrian:~$ kill -SIGQUIT 6578
[1]+  Verlassen                  while true; do
echo -n;
done
```

Generierung bestimmter Signale auch mit der Tastatur möglich:

Signal	*erzeugende Tastaturkombination*
SIGINT	Ctrl-C
SIGQUIT	Ctrl-4 oder Ctrl-\
SIGTSTP	Ctrl-Z

Signale in der Bash

Einrichtung eines Signalhandlers mittels `trap`

```
#!/bin/bash
trap "echo CTRL-C gedrÃ¼ckt. Na fein." SIGINT
trap "echo CTRL-Z gedrÃ¼ckt. Mach ruhig weiter so."
    SIGTSTP
trap "echo Auch SIGQUIT kann mir nix anhaben."
    SIGQUIT
echo Entering loop
while true ; do echo -n ; done
```

Handler wird

- (asynchron) angesprungen,
- ausgeführt, und
- es wird am Unterbrechungspunkt fortgesetzt.

Handler darf nur (externe) Kommandos enthalten, keine Bash-Instruktionen.

Noch ein Bash-Beispiel

`laufschrift.c` (extern)

Probieren Sie:

```
robge@hadrian:~$ ps x | grep  "./laufschrift"
 5341 pts/5    S+     0:00 ./laufschrift
 5343 pts/0    S+     0:00 grep ./laufschrift
robge@hadrian:~$ kill -SIGSTOP 5341
robge@hadrian:~$ kill -SIGCONT 5341
robge@hadrian:~$ kill -SIGSTOP 5341
usw.
```

Signale in C – Teil 1: Senden

```
int kill(pid_t pid, int sig);
```

- sendet das durch `sig` spezifizierte Signal an Prozess mit PID `pid`
- Zuvor wird geprüft, ob der ausführende Nutzer dazu berechtigt ist.
- Spezifikation des Signals: SIGHUP, SIGQUIT, SIGKILL usw., vgl. Headerdatei `bits/signum.h`
- wenn `pid == -1`, dann wird das betreffende Signal an *jeden* Prozess geschickt, für den der Nutzer dieses Recht hat (Vorsicht!)

Was passiert nun bei Zustellung eines Signals?

Behandlung bei Zustellung:

- nichtabfangbares Signal (KILL, STOP) $\rightarrow$ zugeordnete Aktion {Abbruch, Stop} wird ausgeführt
- abfangbares Signal: wenn kein Signalhandler installiert $\rightarrow$ Default-Aktion {Abbruch, Stop, Ignorieren} ausgeführt
- wenn entsprechender *Handler* installiert $\rightarrow$ Handler wird (asynchron zur Programmausführung) aufgerufen

Anmerkungen:

- abfangbares Signal kann auch ignoriert werden

Signale in C – Teil 2: Installation eines Signalhandlers

```
void (*signal(int signum, void (*handler)(int)))(int);
```

fieses Konstrukt; Analyse:

- `signal()` ist ein Systemruf
- übernimmt 2 Parameter:
  - `signum` – Nummer des Signals, für das ein Handler installiert werden soll
  - `handler` – Zeiger auf eine Funktion, die einen Integer übernimmt und nichts zurückliefert
- Rückgabewert: Zeiger auf eine Funktion, die einen Integer übernimmt und nichts zurückliefert (genauso wie `handler()`)

Was bedeutet das?

- Handler ist die Funktion, die angesprungen wird, sobald das entsprechende Signal zugestellt wird
- Parameter des Handlers ist die (konkrete) Nummer des Signals, da es jederzeit möglich ist, einen Handler für verschiedene Signale zu installieren
- Resultat:
  - `SIG_ERR` bei Fehler
  - ansonsten Zeiger auf den vorherigen Handler
- Anstatt des Handlers kann auch übergeben werden:
  - `SIG_IGN` $\rightarrow$ Signal soll ignoriert werden
  - `SIG_DFL` $\rightarrow$ Default-Aktion wird eingestellt.

Beispiel 1: Handler für SIGINT (Ctrl-C)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

long inc = 1;

void ctrl_c_handler (int c)
{
  inc = ( (inc==1) ? -1 : 1);
  return;
}

int main(void)
{
  long count;
  sig_t ret;

  ret = signal(SIGINT, (sig_t) &ctrl_c_handler);
  if (ret == SIG_ERR) {
    perror("signal");
    exit(EXIT_FAILURE);
  }

  /* output count continuously */
  for (count=0; ; count+=inc) {
    printf("%08li\n", count);
  }
  exit(EXIT_SUCCESS);
}
```

(signalhandler.c)

Weitere Beispiele zu Signalen

- Signal bei Ende eines Kindprozesses: `sigchld.c` (extern)
- Redefinition des Handlers im Handler: `catch_ctrl_c.c` (extern)
- Selbstabbruch nach definierter Zeitspanne: `alarm.c` (extern)

Zwischenfazit: Signale unter POSIX

4 „klassische“ Funktionen:

- `kill()`
- `signal()`
- `pause()` - wartet (passiv) auf ein Signal
- `alarm()` - definiert eine Zeitspanne, bis `SIGALRM` zugestellt wird („Der Wecker klingelt.“)

Nachteile und Unzulänglichkeiten von Signalen

- unzuverlässig
- keine Nutzdatenübertragung
- keine Priorisierung
- keine Speicherung (in Warteschlange)

modernere (aber kompliziertere) Signalbehandlung:
`sigaction()`, `sigprocmask()` & Co.

Gemeinsam genutzter Speicher (*Shared Memory*)

- Idee: Kommunikation über gemeinsamen Speicher
- keine implizite Synchronisation (!)
- ohne Adressräume problemlos implementierbar
- bei virtuellem Speicher Funktionen des BS nötig:
  - Anlegen des Segmentes
  - Einblenden in beide Adressräume
  - (Datenübertragung)
  - Ausblenden aus allen Adressräumen
  - Zerstören des Segments
- Zugriff auf gemeinsam genutzten Speicher über Zeiger, überlagerte Datenstrukturen ($\rightarrow$ effizient), kein Systemruf nötig
- UNIX: `shmget()`, `shmat()`, `shmdt()`, `shmctl()`
- Segmente sind i.a. persistent (überleben den anlegenden Prozess)

Nachrichtenaustausch (Message Passing)

Prinzip

1. Sender konstruiert Nachricht und trägt diese in einen Puffer ein
2. Sender ruft Funktion `send()`
3. Nachricht wird durch das System transportiert
4. Empfänger ruft Funktion `receive()`, der er einen Puffer übergibt, in den die Nachricht kopiert wird

Analogie: Briefsendung

Nachrichtenaustausch (Message Passing)

Diskussion

- notwendig, wenn kein gemeinsamer Speicher existiert (z. B. in verteilten Systemen)
- jedoch auch mit gemeinsamem Speicher möglich (z. B. Unix)
- zwei grundlegende Operationen: `send()`, `receive()`
- synchrone und asynchrone Operation möglich

Beispiele:

- Message Passing Interface (MPI)
- Nachrichtenwarteschlangen POSIX (`msgrcv()`, `msgsnd()` usw.)

Synchroner und asynchroner Nachrichtenaustausch

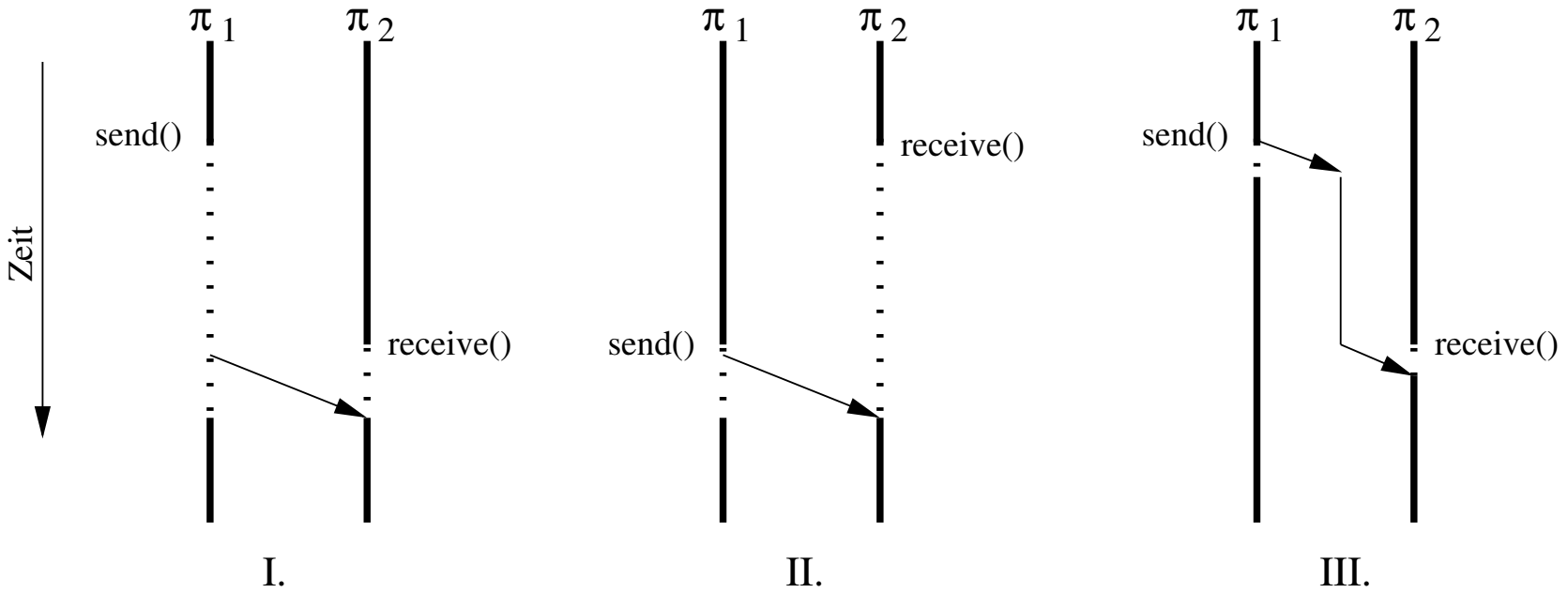


Abbildung: I./II. – blockierend, III. nichtblockierend (asynchron)

Kommunikation über Sockets

. . . in der LV „Rechnernetze“

Was haben wir gelernt?

1. Was sind IPC-Mechanismen?
2. ausführlich kennengelernt:
   - Datenaustausch mittels Datei
   - die „klassische“ Unix-Pipe
   - Signale (zumindest in der Bash)
3. kurz angerissen
   - *Shared Memory*
   - *Message Passing*
4. nicht behandelt
   - *Sockets*
   - *named Pipes*

Testfragen

1. Beschreiben Sie den Ablauf bei der Zustellung eines Signals!
2. Was ist der Unterschied zwischen synchronem und asynchronem Senden beim Message Passing?
3. Entwerfen Sie ein C-Programm, das einen Sohn erzeugt und diesem mittels einer Pipe eine Nachricht zukommen lässt!
4. Welchen Kommunikationsmechanismus würden Sie einsetzen, wenn Sie Daten übertragen müssen und die Teilnehmer nicht verwandt sind?
5. Was ist ein *persistenter* Kommunikationsmechanismus?