

# Interprozesskommunikation (IPC)

Vorlesung Betriebssysteme - 12.12.2018

Carsten Hahn

## Interprozesskommunikation (IPC)

- Motivation
- Beispiele
- Klassifikation
- Mechanismen

### IPC zur **Synchronisation**

- Lock Files
- File-Lock
- Signale

### IPC zur **Kommunikation**

- Message Queues
- Pipes
- Shared Virtual Memory
- Mapped Memory
- Distributed Virtual Memory
- Message Passing
- Sockets
- RPC



LUDWIG-  
MAXIMILIANS-  
UNIVERSITÄT  
MÜNCHEN

## Teil 1

 mobile and  
distributed systems group



# Motivation und Beispiele

- Verschiedene Prozesse wollen miteinander kommunizieren
  - Austausch von Daten zwischen Prozessen (z.B. auch Signale)
  - Synchronisation von Zugriffen auf Ressourcen
    - gemeinsamer Dateizugriff
    - Zugriff auf Systemressourcen, Geräte usw.
    - ...
- Beispiele
  - Erfüllung gemeinsamer Aufgabe
  - Weiterleitung von Ereignissen/Nachrichten
  - Erzeuger-Verbraucher Problem
  - Prozesssynchronisation
  - ...

- Erfüllung gemeinsamer Aufgabe
  - Aufteilung komplexer mathematischer Berechnungen auf mehrere Prozessoren auf einem oder mehreren Rechnern
  - Beispiel: Parallele Algorithmen für die Matrixmultiplikation

- Sequentielle Multiplikation

$$C = A \cdot B \quad A, B, C \in R^{2^k \times 2^k} \quad \Rightarrow \quad n^3 = n^{\log_2 8}$$

- Zerlegung in Blöcke

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \quad A_{ij}, B_{ij}, C_{ij} \in R^{2^{k-1} \times 2^{k-1}}$$

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

$$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

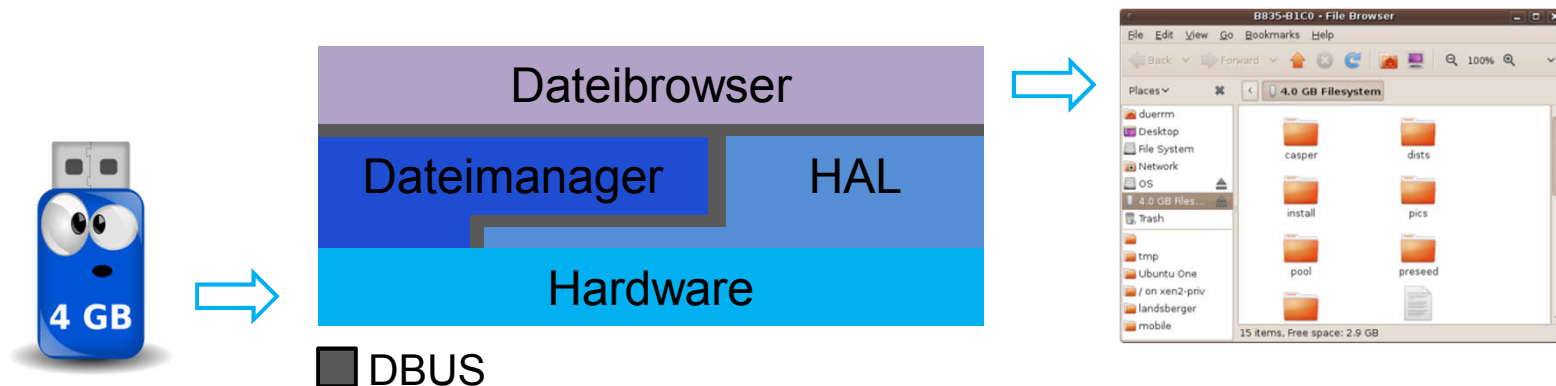
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

$$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

$\Rightarrow$  Auf Multicore Architektur bis zu vierfacher Beschleunigung

- Weitere Optimierungen möglich
  - (Strassen Algorithmus  $\Rightarrow$  Nutzt Hilfsmatrizen )

- Weiterleitung von Ereignissen/Nachrichten
  - Kommunikation von Hintergrundprozessen (Daemons) mit (Desktop-) Anwendungen
  - Beispiel: Kommunikation des HAL-Daemon (Hardware Abstraction Layer Daemon) mit dem Dateimanager :
    - Automatisches mounten eines USB-Sticks
      - Kommunikation über Dbus System (Ereignisse als Nachrichten)
      - HAL-Daemon informiert Dateimanager über Hardware-Änderung
      - Dateimanager mountet automatisch USB-Stick und zeigt Inhalte an.



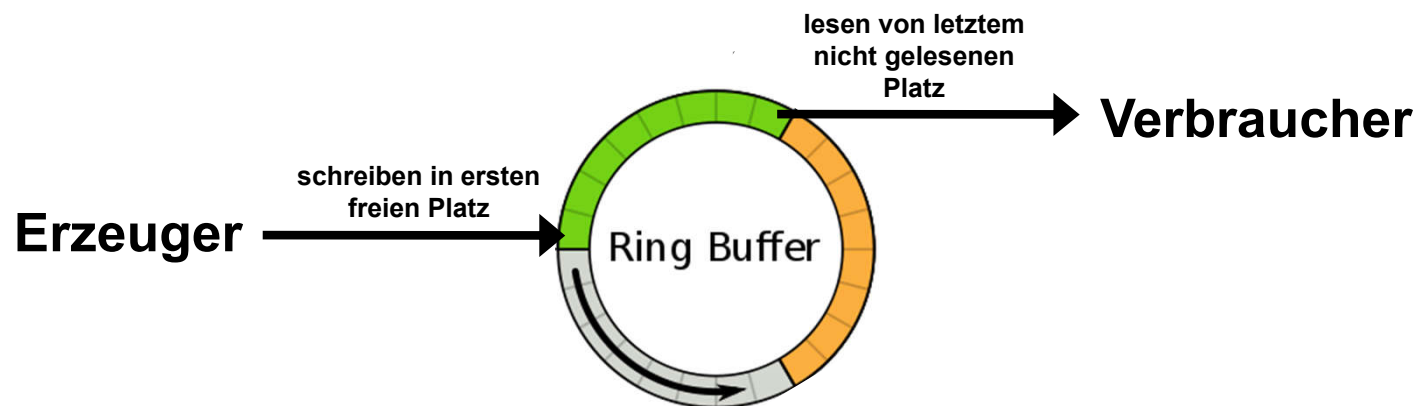
- Pipelining
  - Interaktion modularer Programme (Programmteile) über Pipelines
  - Beispiel: Koppelung von Systemprogrammen mit Unix-Pipes
    - Prozess A generiert Daten, die ein Prozess B als Eingabe benötigt

```
host$ cat /etc/passwd | grep kennung | cut -d":" -f5
```



liefert den Namen der Person mit Kennung **kennung**

- Prozesssynchronisation
  - Zugriff auf gemeinsam genutzten Speicher muss synchronisiert erfolgen
  - Beispiel: Erzeuger-Verbraucher-Problem
    - Prozesse schreiben Daten in einen Puffer (z.B. Ring Puffer im Shared Memory), die andere Prozesse auslesen.



- Lesen und Schreiben nur im **wechselseitigen Ausschluss** erlaubt!
- Sonst können **Race-Conditions** auftreten



- Race Condition (Kritischer Wettlauf)
  - Definition:
    - Eine Race Condition entspricht einer Situation, in der das Ergebnis einer Abfolge von Operation vom zeitlichen Verhalten bestimmter Einzelinstruktionen abhängt.
  - Problem:
    - Ergebnis ist nicht-deterministisch (abhängig von der Reihenfolge der Prozessausführung)
  - Ursachen:
    - Unterbrechung durch blockierende Funktion, Hardware Interrupts, Multiprocessing,...
  - Beispiel:
    - Paralleler Zugriff auf einelementigen Puffer X:
      - Prozess A schreibt in X, Prozess B liest aus X
    - Bedingung: B darf erst lesen, wenn A geschrieben hat
      - Erlaubter Ablauf:  $A \rightarrow B \rightarrow A \rightarrow B \rightarrow \dots$
      - Illegaler Ablauf:  $A \rightarrow B \rightarrow B \rightarrow A \rightarrow \dots$

➡ Der einelementige Puffer stellt einen **kritischen Bereich** dar

- Kritischer Bereich

- Definition:

- Als **kritischen Bereich** bezeichnet man ein Stück Programmtext, innerhalb dessen auf ein gemeinsames Betriebsmittel (Datenstruktur oder Gerät) zugegriffen wird und auf welches nicht von mehr als einem Prozess oder Thread zum selben Zeitpunkt zugegriffen werden darf.

- Bedeutung:

- Kritischer Bereich muss geschützt werden um Race Condition zu vermeiden

- Wie kann der Kritische Bereich geschützt werden?

- Verwendung von **Semaphoren** oder **Monitoren**

- Semaphore



- Definition:

- Ein **Semaphor** bezeichnet eine geschützte Variable, die eine einfache Zugriffskontrolle mehrerer Prozesse oder Threads auf ein gemeinsames Betriebsmittel realisiert (**wechselseitiger Ausschluss**)

- Umsetzung:

- Integer-Variable  $S$  auf die drei atomare Operationen anwendbar sind
  - **init**( $S$ ) setzt  $S$  auf seinen Anfangswert
  - **wait**( $S$ ) versucht  $S$  zu dekrementieren
  - **signal**( $S$ ) inkrementiert  $S$

- Ablauf:

- Prozess  $P$  will in Kritischen Bereich   $P$  versucht  $S$  zu dekrementieren
  - $S > 0$   $P$  darf in den Kritischen Bereich
  - $S \leq 0$   $P$  kommt in eine Warteschlange
- $P$  verlässt Kritischen Bereich   $P$  inkrementiert  $S$  und benachrichtigt wartende Prozesse

- Semaphor

- Beispiel:

- Zugriff auf einelementigen Puffer X durch Prozesse A und B (ohne Berücksichtigung der Zugriffsreihenfolge!)

- Schritt 1: Initialisierung des Semaphor **mutex**

```
init(mutex, 1) // hier mutex binäre Semaphore
```

- Schritt 2: Anwendung im Programm

Prozess A	Prozess B
-----	-----
...	...
wait(mutex)	wait(mutex)
<krit. Bereich A>	<krit. Bereich B>
signal(mutex)	signal(mutex)
...	...

- Anmerkung:

- neben der **binären Semaphore** gibt es auch die **Zähl-Semaphore**
- Mehr zu **Semaphoren** und **Monitoren** in einer der nächsten Vorlesungen

- Lokale und verteilte IPC
  - Historisch bedingte Unterscheidung:
    - Zuerst IPC nur auf Einprozessorsystemen
    - Hier nur sehr eingeschränkte Anforderungen an IPC
  - Lokale IPC
    - Kommunikation basiert auf gemeinsamen Speicherbereich
    - Beispiele:
      - Lesen und Schreiben von Shared Memory
      - Synchronisation über Datei-Lock
  - Verteilte IPC
    - Kein gemeinsamer Speicherbereich vorhanden
    - Kommunikation über Systemgrenzen hinweg
    - Beispiel:
      - Kommunikation über Sockets

⇒ Verteilte IPC bedingt Einführung anderer Techniken

Methode	Unterstützung
Semaphore	Alle POSIX Systeme
Monitore	Viele Programmiersprachen
Datei / Datei Lock	Fast alle BS
Signale	Fast alle BS
Message Queues	Fast alle BS
(Stream) Pipe	Alle POSIX Systeme
Named (Stream) Pipe	Alle POSIX Systeme, Windows
Shared Virtual Memory	Alle POSIX Systeme
Memory Mapped Datei	Alle POSIX Systeme, Windows
Sockets	Fast alle BS
Message Passing (kein Shared Memory)	RPC, JavaRMI, CORBA, Objective C, ...
Distributed Shared Memory	Spezielle System Clustering Software

POSIX = Portable Operating System Interface (IEEE standardisierte API zw. Nutzerprogrammen und BS)



LUDWIG-  
MAXIMILIANS-  
UNIVERSITÄT  
MÜNCHEN

## Teil 2

 mobile and  
distributed systems group



# IPC zur Synchronisation

- Hatten wir schon...
  - Problem:
    - Prozesse greifen auf gemeinsame Ressource zu
      - ⇒ Race Condition möglich
      - ⇒ Synchronisation notwendig
    - Beispiel: Zwei Prozesse wollen in dieselbe Datei schreiben (Logging)
  - Lösung:
    - Synchronisation über *Semaphore* und *Monitore*
      - Dazu gibt es eine gesonderte Vorlesung...
- Weitere Lösungen?



- Lösung mit Lock-File (Sperrdatei)

- Einfacher Ansatz:

- Synchronisation über einfache Sperrdatei

- Vorgehensweise:

- Prozess versucht vor Schreibzugriff Lock-Datei zu erzeugen
  - Meist im Verzeichnis */tmp*
  - Nur dieser Prozess hat Schreibrechte!
- Wenn Datei schon vorhanden
  - ⇒ Versuch schlägt fehl
  - ⇒ Erneuter Versuch nach best. Zeit: *sleep()*
- Bei Erfolg
  - ⇒ krit. Bereich betreten, Code ausführen, krit. Bereich verlassen
  - ⇒ Sperrdatei freigeben: *unlink()*



## Probleme:

- Namenskonvention für Lock-Datei
- Busy Waiting
- Keine Scheduling Strategie  $\Rightarrow$  Prozess kann auch verhungern
- Lock-Datei selbst ist eine gemeinsam genutzte Ressource  
 $\Rightarrow$  Atomarer Zugriff notwendig!
- Superuser hat immer Schreibrechte  
 $\Rightarrow$  Könnte in die Routine eingreifen

- Lösung mit File-Lock (Dateisperre)
  - Vorgehensweise:
    - Lock auf Datei selbst, die verändert werden soll
  - Unterscheidung zwischen
    - Pflichtsperre (Mandatory)
      - Zugriffsrechte einer Datei bei Zugriff beachten!
    - Kooperationssperren (Advisory)
      - Funktionsbibliothek mit Zugriffsfunktionen
  - Beispiel: UNIX
    - „...*UNIX locks are advisory not mandatory*...“
      - Prozesse **dürfen** Datei-Locks für synchronisierten Zugriff auf Datei verwenden
      - Prozesse **können** Datei-Locks aber auch gänzlich ignorieren.
    - Shared Locks (Read-Locks) vs. Exclusive Locks (Write-Locks)

- Shared Locks
  - Unbegrenzte Anzahl von Prozessen
  - In Koexistenz mit mehreren Shared Locks anwendbar
  - Falls Exclusive Lock existiert
    - ⇒ Warten bis Exclusive Lock freigegeben wurden
- ⇒ Lesen ist sonst immer erlaubt (unkritisch)
- Exclusive Locks
  - Genau ein Prozess
  - Nicht in Koexistenz mit weiteren Locks (Shared oder Exclusive) anwendbar
  - Falls irgendein Lock existiert
    - ⇒ Warten bis alle Locks freigegeben wurden

- Lösung mit File-Lock

- Beispiel: UNIX

- Systemfunktionen `fcntl()` und `flock()`

- `fcntl()`

- Locking auf einzelne Byte-Bereiche einer Datei
        - *Advisory* Locking
        - ***Mandatory*** Locking möglich (Unterstützung durch das Dateisystem notwendig!)

- `flock()`

- Locking auf gesamte Datei
        - Nur ***Advisory*** Locking
        - `flock` auch als Kommando zur Shell-Programmierung verfügbar (`man flock`)

- Probleme:

- Besonderheiten bei der Verwendung beider Funktionen abhängig von der jeweiligen UNIX Implementierung (Sperrarten nicht genormt!)
    - Mandatory Locks nur sinnvoll, wenn alle Prozesse sich an Locking Regeln halten.
    - Nutzer kann absichtlich Exclusive Lock setzen und halten, so dass andere Anwender keinen Zugriff mehr erhalten

- Lösung mit Signalen
  - Allgemeines:
    - Einer der ältesten IPC-Mechanismen
    - Software Mechanismus zur Benachrichtigung eines Prozesses über *asynchrone* Ereignisse
    - Signale ähnlich zu Hardware Interrupts, **aber**
      - besitzen keine Priorität
      - werden gleichberechtigt behandelt
    - Kategorien:
      - Systemsignale (Hardware- oder Systemfehler, SIGKILL, SIGTRAP, ...)
      - Gerätesignale (SIGHUP, SIGINT, SIGSTOP, SIGIO, ...)
      - Benutzerdefinierte Signale (SIGQUIT, SIGABRT, SIGTERM, SIGUSR, ...)
  - Auflistung der Signale mit ***kill -l***

Name	Wert	Aktion	Bemerkung
<b>SIGINT</b>	2A		Interrupt-Signal von der Tastatur
<b>SIGQUIT</b>	3A		Quit-Signal von der Tastatur
<b>SIGILL</b>	4A		Falsche Instruktion
<b>SIGTRAP</b>	5CG		Überwachung/Stopp-Punkt
<b>SIGABRT</b>	6C		Abbruch
<b>SIGUNUSED</b>	7AG		Nicht verwendet
<b>SIGFPE</b>	8C		Fließkomma-Überschreitung
<b>SIGKILL</b>	9AEF		Beendigungssignal
<b>SIGUSR1</b>	10A		Benutzer-definiertes Signal 1
<b>SIGSEGV</b>	11C		Ungültige Speicherreferenz
<b>SIGUSR2</b>	12A		Benutzer-definiertes Signal 2
<b>SIGPIPE</b>	13A		Schreiben in Pipeline ohne Lesen
<b>SIGALRM</b>	14A		Zeitsignal von alarm(1)
<b>SIGTERM</b>	15A		Beendigungssignal
<b>SIGSTKFLT</b>	16AG		Stack-Fehler im Coprozessor
<b>SIGCHLD</b>	17B		Kindprozess beendet
<b>SIGCONT</b>	18		Weiterfahren, wenn gestoppt
<b>SIGSTOP</b>	19DEF		Prozessstopp
<b>SIGTSTP</b>	20D		Stopp getippt an einem TTY
<b>SIGTTIN</b>	21D		TTY-Eingabe für Hintergrundprozesse

Name	Wert	Aktion	Bemerkung
<b>SIGTTOU</b>	22D		TTY-Ausgabe für Hintergrundprozesse
<b>SIGIO</b>	23AG		E/A-Fehler
<b>SIGXCPU</b>	24AG		CPU-Zeitlimite überschritten
<b>SIGXFSZ</b>	25AG		Dateien Größenlimite überschritten
<b>SIGVTALRM</b>	26AG		Virtueller Zeitalarm
<b>SIGPROF</b>	27AG		Profile Signal
<b>SIGWINCH</b>	29BG		Fenstergrößenänderung

## Linux Signale

Die Zeichen in der Spalte "Aktion" haben folgende Bedeutung:

- A:** Normalerweise wird der Prozess abgebrochen.
- B:** Normalerweise wird dieses Signal ignoriert.
- C:** Normalerweise wird ein "Core Dump" durchgeführt.
- D:** Normalerweise wird der Prozess gestoppt.
- E:** Signal kann nicht abgefangen werden.
- F:** Signal kann nicht ignoriert werden.
- G:** Kein POSIX.1-konformes Signal.

- Lösung mit Signalen
  - Umsetzung (I):
    - Jeder Prozess besitzt Signal Maske (32 Bit Adressbreite => 32 Signale)
    - Kein Signal-Queuing
      - ⇒ Auftreten mehrerer Signale des gleichen Typs nicht erkennbar!
    - Signale nur im Nutzermodus behandelbar
      - Signal während Prozess im Kernelmodus
        - ⇒ Sofortige Rückkehr in Nutzermodus
        - ⇒ Abarbeitung des Signals
  - Generell:
    - Signal tritt auf ⇒ Eintrag in Prozesstabelle (Pending Signal)
    - Behandlung sobald Prozess aufwacht oder von Systemaufruf zurückkehrt



- Signale können auftreten
  - Durch Programmfehler (Bsp.: SIGFPE  $\Rightarrow$  Division durch Null)
  - Durch Benutzer selbst
    - » Bsp.: **STR+C**  $\Rightarrow$  SIGINT (Beendigung)
    - » Bsp.: **STR+Z**  $\Rightarrow$  SIGTSTP (Anhalten)
- Prozesse können:
  - Signale blocken (ignorieren)
  - Signal-Handler für bestimmte Signale installieren (Handler-Funktion)
  - dem Kernel Signalbehandlung überlassen (Standardaktion)

- Lösung mit Signalen

- Umsetzung (II):

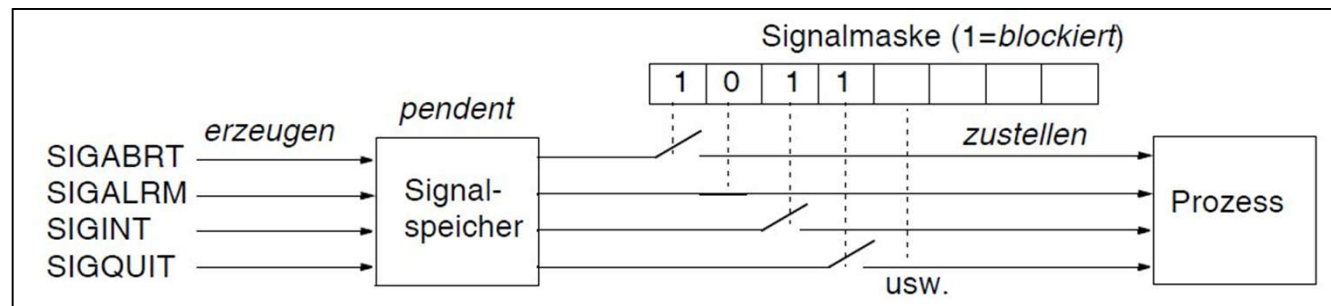
- Signale blocken

- Geblocktes Signal tritt ein

⇒ Signal **"hängt" (pending)** bis es deblockiert wird

- Hängende Signale können später noch erkannt werden

- Mit **SIGKILL** und **SIGSTOP** nicht möglich!



- Signal-Handler

- Signal tritt ein ⇒ Handler-Funktion wird ausgeführt

- Signal Behandlung durch Kernel

⇒ Kern initiiert Default-Behandlung  
(z.B. **SIGFPE** – Division durch Null ⇒ Core Dump + Abort)

- Lösung mit Signalen

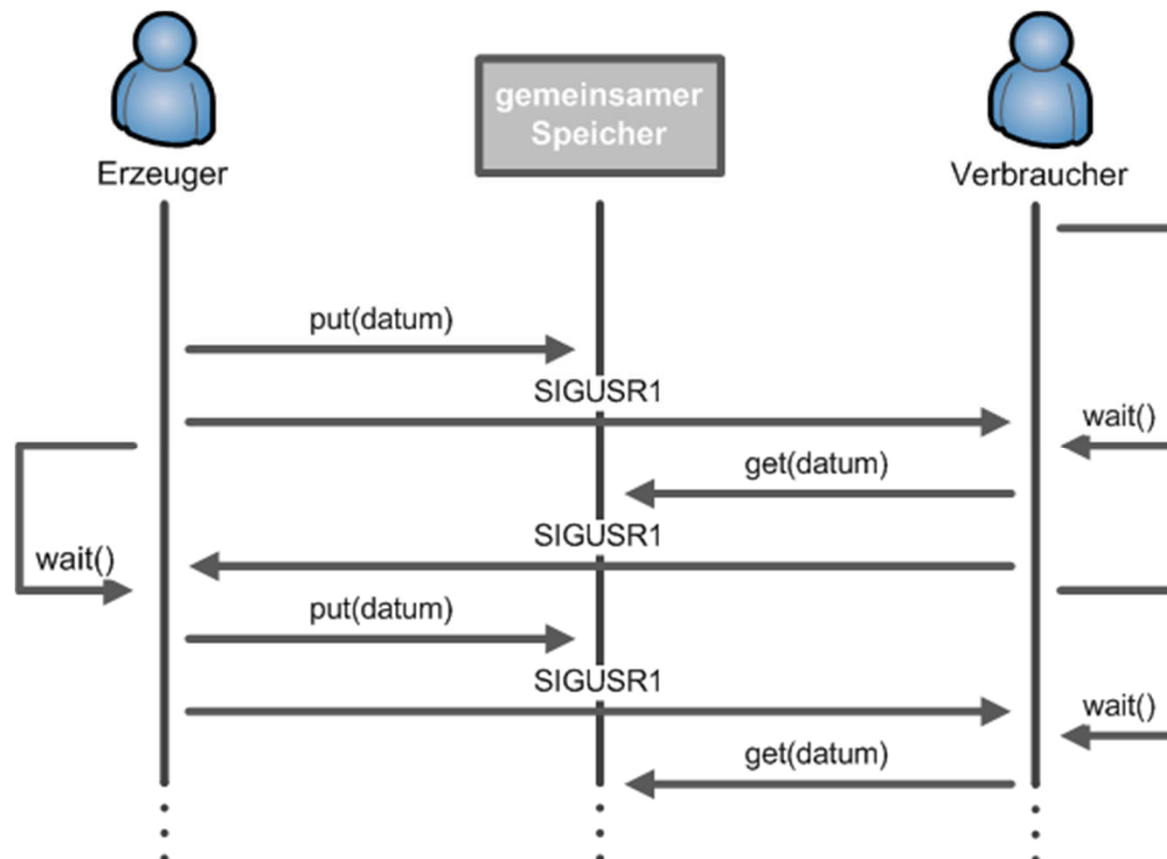
- Verwendung zur IPC:

- Unter Linux: Benutzerdefinierbare Signale **SIGUSER1** und **SIGUSER2**
    - Verwendung der `wait()`-Funktion um auf Unterbrechung durch Signal zu warten (kein Busy Waiting!)
    - Signal tritt ein
      - ➡ Versetze wartenden Prozess in Zustand **"Running"**
      - ➡ Ausführen des Signal-Handlers

- Achtung:

- Signale Senden geht nur zwischen Prozessen mit derselben
      - User ID,
      - Group ID, oder
      - Prozessgruppe
    - Ausnahme: Kernel
      - ➡ Kann Signale an X-beliebige Prozesse versenden

- Lösung mit Signalen
  - Beispiel: Erzeuger/Verbraucher Problem



## Interprozesskommunikation (IPC) zur Synchronisation:

- Ziel: Zugriffskontrolle (Vermeidung von Race Conditions)
  - Lock-Files
    - Steuert Zugriff auf kritischen Bereich
    - Nachteile: Busy Waiting, Namenskonvention, Prozesse können verhungern, Superuser hat immer Schreibrechte
  - File-Lock
    - Steuert Zugriff auf die Datei selbst
    - Mandatory bzw. Advisory
    - Nachteile: Nicht standardisiert, Absichtliches Halten von Exclusive Lock
  - Signale
    - Asynchrone Ereignisse (Interrupts)
    - Blockieren, Behandeln durch Prozess, Standardbehandlung durch Kernel



LUDWIG-  
MAXIMILIANS-  
UNIVERSITÄT  
MÜNCHEN

## Teil 3

 mobile and  
distributed systems group



# IPC zur Kommunikation

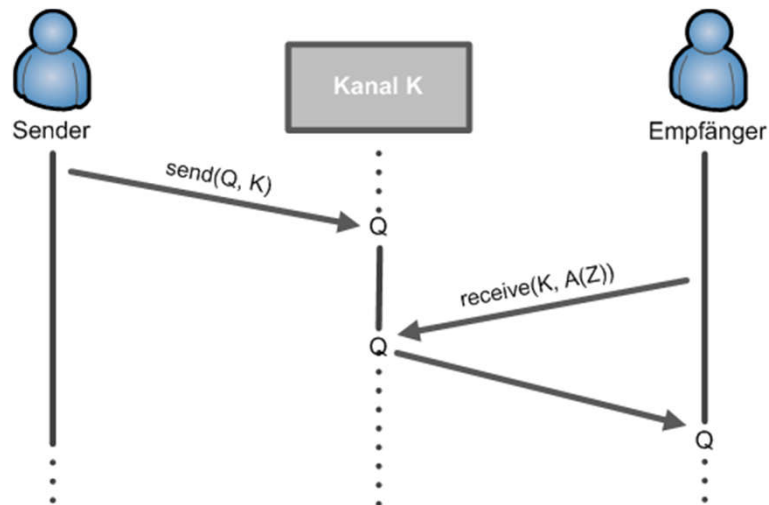
- Motivation:
    - Prozesse wollen Daten untereinander kommunizieren
  - Historie:
    - In klassischen Betriebssystemen stark vernachlässigt
    - In verteilten Systemen kein gemeinsam genutzter Speicher vorhanden
    - Zwang die Arbeit auch über Rechengrenzen hinweg zu koordinieren
- ⇒ IPC wird auch auf einzelnen Rechnern als Betriebssystemdienst ermöglicht

- Abstrakte Sichtweise
  - Erzeugung eines Kanals zwischen zwei (oder mehreren) Prozessen
  - Prozess A
    - Schreibt Datenstrom (Bytes) von seinem Quellbereich in gesonderten Speicherbereich **K** (Kanal)
  - Prozess B
    - Liest Datenstrom aus **K** und schreibt Daten in seinen Zielbereich
  - Operationen:
    - **send**(Quelldaten, Kanal)
    - **receive**(Kanal, Zieldatenbereich)
  - Unterscheidung
    - *Asynchrone (nicht-blockierende) Kommunikation*
    - *Synchrone (blockierende) Kommunikation*

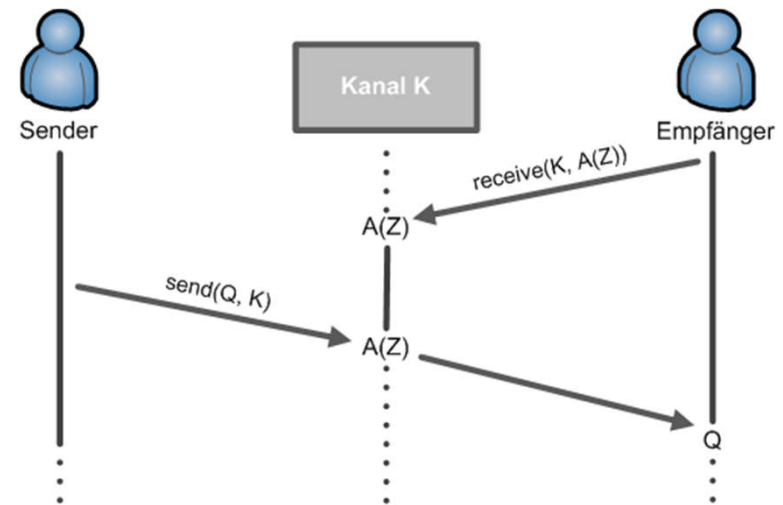


- Asynchrone (nicht-blockierende) Kommunikation
  - Zeitpunkt der Abholung / des Empfangs der Daten für Sender- bzw. Empfängerprozess irrelevant

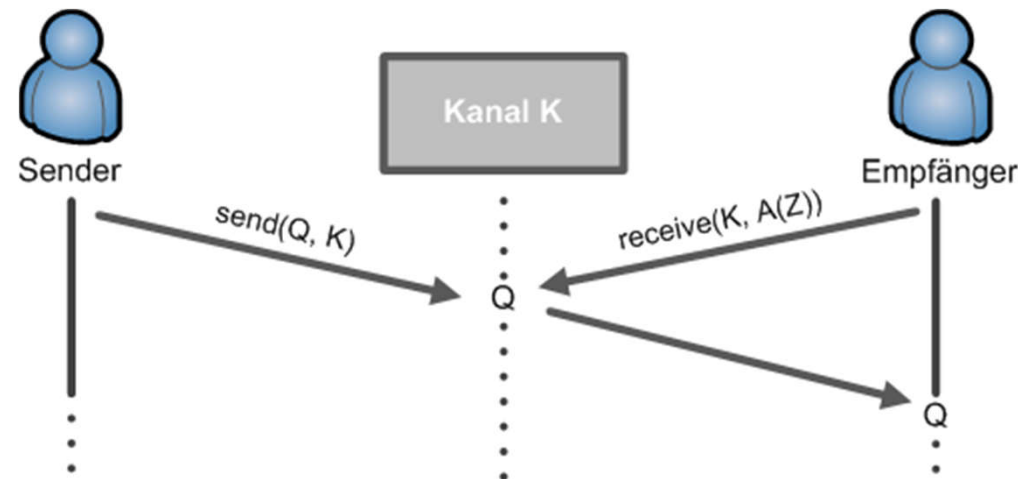
- Aufruf von **send()** vor **receive()**
  - Zwischenspeichern der **Daten** im Kanal bis **receive()** erfolgt



- Aufruf von **receive()** vor **send()**
  - Zwischenspeichern der **Adresse** des Zieldatenbereichs im Kanal (Pointer auf Adressbereich)



- Synchrone (blockierende) Kommunikation
  - Rendezvous (zeitliche Abstimmung zwischen Sender und Empfänger)
    - Sender weiß dass Empfänger Daten erwartet
    - Empfänger weiß, dass Sender Daten senden will

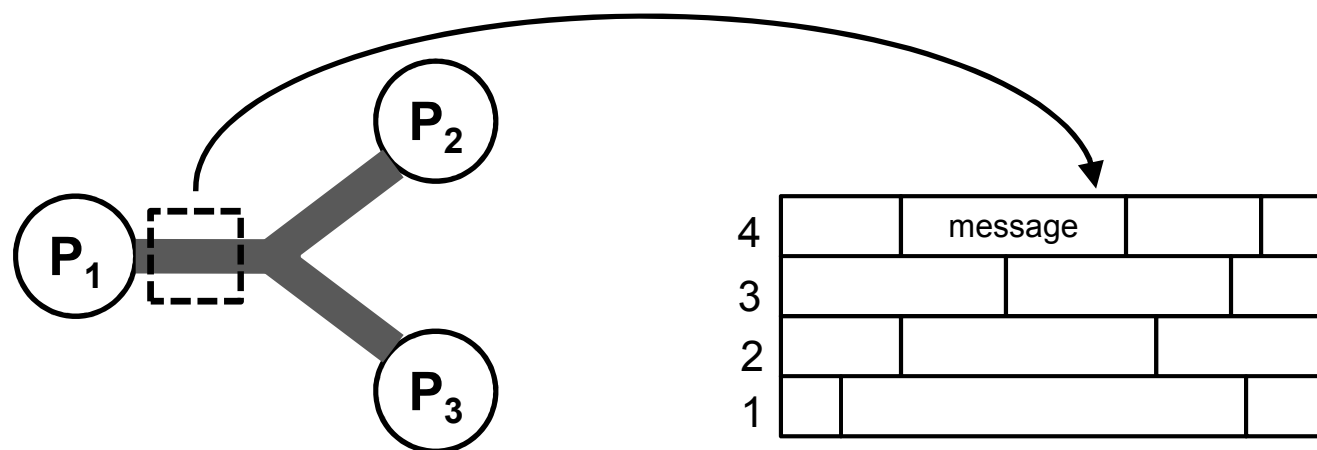


- Anmerkung
  - Bei synchroner Kommunikation können **Deadlocks** auftreten
  - Beispiel: Anforderung einer Website bei Single-Threaded Webserver

- Grundlagen
  - Drei verschiedene Arten des Nachrichtenaustauschs
    - Unicast (Punkt zu Punkt)
    - Multicast (Punkt zu Mehrpunkt)
    - Broadcast (Rundsendung)
  - Arten der Kommunikation
    - **Verbindungsorientiert**
      - Vereinbarung über Kommunikationsablauf zu Beginn (Empfänger existiert und kann Verbindung annehmen  $\Rightarrow$  Verbindung wird aufgebaut)
      - **Vorteil:** Nachrichten können nicht verloren gehen
      - **Nachteil:** Verwaltung von Zuständen auf Sender- und Empfängerseite
      - Beispiele: Telefonnetz, TCP-Protokoll, SSH-, HTTP-Sessions, ...
    - **Verbindungslos**
      - Keine Vereinbarung über Kommunikationsablauf
      - **Vorteil:** Keine Zustandsinformationen notwendig
      - **Nachteil:** Nachrichtenverlust möglich
      - Beispiel: Briefpost, IPv4/6-, UDP-Protokoll, Video-On-Demand

- Message Queues

- Ein oder mehrere (unabhängige) Prozesse generieren Nachrichten, die von einem oder mehreren Prozessen gelesen werden können.
- Abarbeitung erfolgt nach FIFO-Prinzip, **aber** selektives Lesen erlaubt!
- Sender und Empfänger müssen nicht gleichzeitig laufen (asynchron)
  - Beispiel:
    - Sender kann Message Queue öffnen, hineinschreiben und sich wieder beenden
    - Leser kann dennoch erst jetzt Nachrichten aus der Message Queue lesen

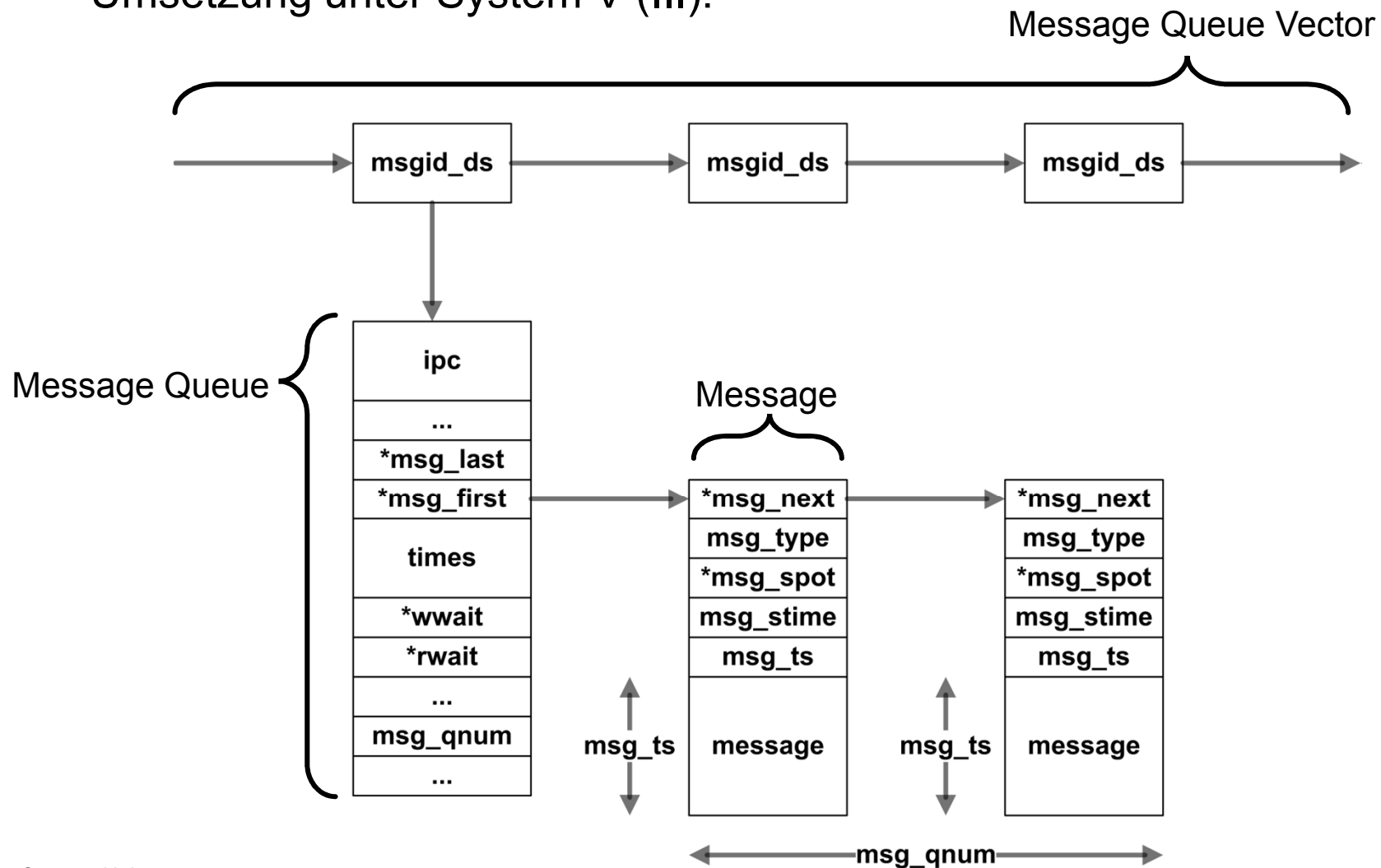


- Umsetzung unter System V\* (I):
  - Prozesse können
    - Message Queues erzeugen
    - sich mit existierender Message Queue verbinden
  - Voraussetzung zum Erzeugen / Verbinden:
    - Angabe eines Schlüssels für die Message Queue
    - **Problem:** Prozesse müssen vor Ausführung
      - den Schlüssel, oder
      - dessen Erzeugungsmechanismus kennen
  - Bei Erzeugen / Verbindung kann Prozess Zugriffsrechte angeben
    - read-write, read-only, ...

\*Klasse von Unix-Derivaten

- Umsetzung unter System V (II):
  - Jegliche Art von Datenstruktur als Nachricht möglich
    - Einschränkung: erstes Element einer Message vom Datentyp `long`
      - Gibt den Nachrichtentyp an
      - Ermöglicht Auslesen des ersten Elements bzw.
      - Ermöglicht Auslesen aller Elemente mit entspr. Datentyp
  - Kernel unterhält Liste von Message Queues (Message Queue Vector)
    - Neue Queue wird an Vektor angehängt
    - Listenelemente verweisen auf Verwaltungsdatenstruktur einer Queue (Modifikationszeit, wartende Prozesse, ...)
    - Message Queue selbst als Liste organisiert
      - Neue Nachrichten werden am Ende der Liste angehängt (**Kopieren** vom Nutzeradressraum in Kerneladressraum)
      - Überprüfung der Zugriffsrechte bevor Nachricht angehängt wird
      - Queues haben **limitierte Größe** (Write-Wait-Queue und Read-Wait-Queue falls Schreiben/Lesen nicht möglich)

- Umsetzung unter System V (III):



- (Unnamed) Pipes
  - Erste Form der IPC, heute gängig in allen UNIX-Systemen
  - Spezieller gepufferter Kanal
  - Unidirektionaler Byte-Strom
    - Verknüpft Standardausgabe von Prozess A mit Standardeingabe von Prozess B
  - Halb-Duplex => Kommunikation nur unidirektional (Von A nach B)
  - Pipes nur zwischen Prozessen mit gemeinsamen Vorfahren
    - (Groß-)Elternprozess richtet Pipe ein, Kindprozesse **erben** Pipe  
⇒ Kommunikation zwischen allen möglich
  - Prozesse bekommen nichts von der "Umleitung" mit
    - Kernel kümmert sich um Synchronisation (Read-, Write-Locks)



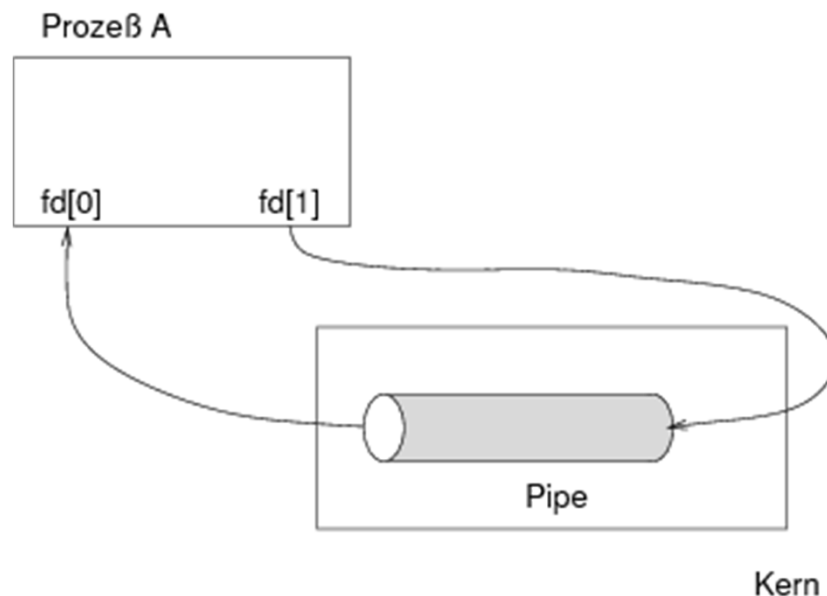
- Beispiel: Unix Shell
  - Koppelung von Systemprogrammen

```
host$ cat /etc/passwd | grep kennung | cut -d":" -f5
```

Es passiert folgendes:

- Shell startet Erzeugung von drei Prozessen: `cat`, `grep`, `cut`
  - Ausgabe von `cat` dient als Eingabe von `grep`
  - Ausgabe von `grep` dient als Eingabe von `cut`
  - Ausgabe von `cut` wird auf die Standardausgabe (shell) geschrieben
- In Mehrprozessorumgebung echt parallele Ausführung

- Beispiel: Linux **pipe()** Funktion (I)
  - Pipe einrichten über Systemaufruf **pipe()**  
⇒ Zwei Dateideskriptoren **fd[0]** (Lesen) und **fd[1]** (Schreiben)

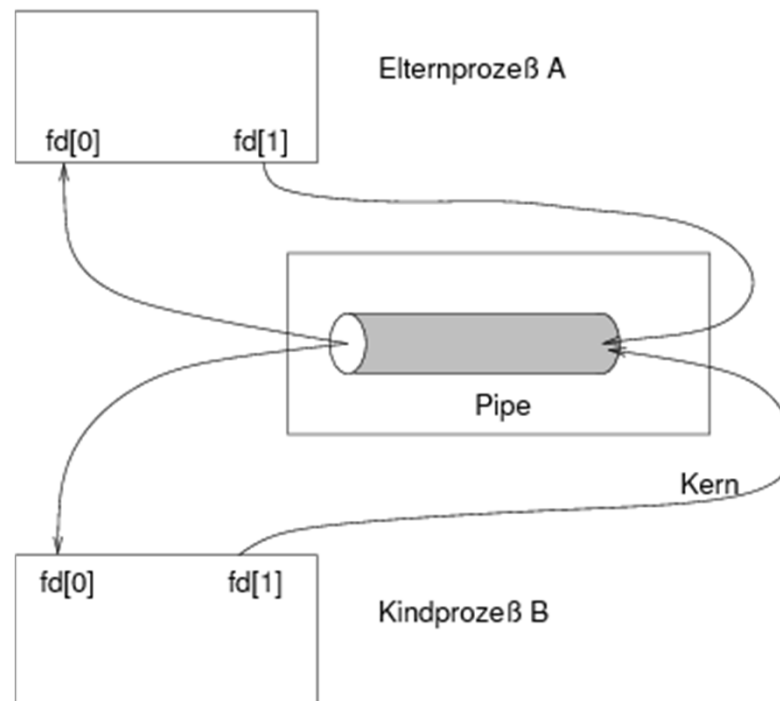


- Was nutzt uns das jetzt...?

- Beispiel: Linux **pipe()** Funktion (II)
  - Erstellen eines Kindprozesses (**fork**)

⇒ 1:1 Kopie

- **Achtung:** Gemeinsam genutzte Ressourcen für IPC werden **nicht kopiert** sondern **verlinkt**!

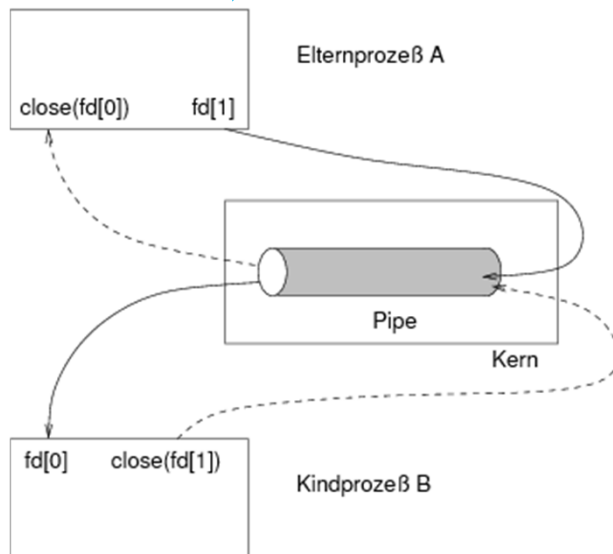


- Zwei Möglichkeiten der Kommunikation
  - Parent ⇒ Child
  - Child ⇒ Parent
- Und was nutzt und das jetzt...?

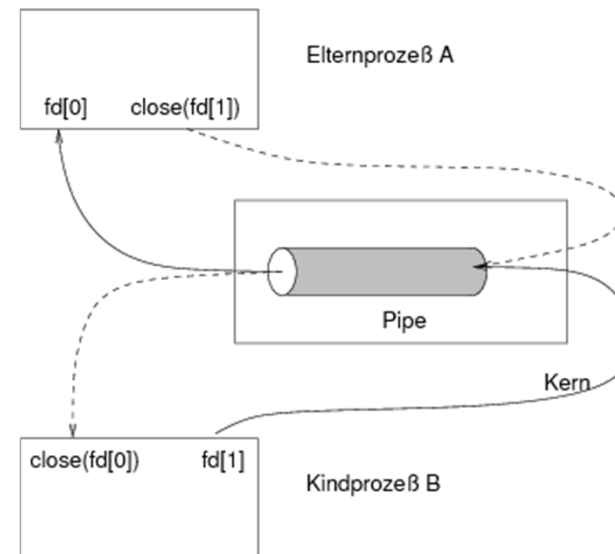
- Beispiel: Linux **pipe()** Funktion (III)

- Schließe nicht verwendete Dateideskriptoren

- Parent  $\Rightarrow$  Child



- Child  $\Rightarrow$  Parent



- Anmerkungen:

- Paralleles Schreiben mehrerer Prozesse in eine Pipe

- $\Rightarrow$  Daten werden sequentiell abgelegt

- Prozess schreibt mehr Daten in einen Pipe Puffer als Größe der Pipe zulässt

- $\Rightarrow$  Prozess wird blockiert bis Pipe Puffer geleert

- Beispiel: Kommunikation zwischen Kind- und Vaterprozess

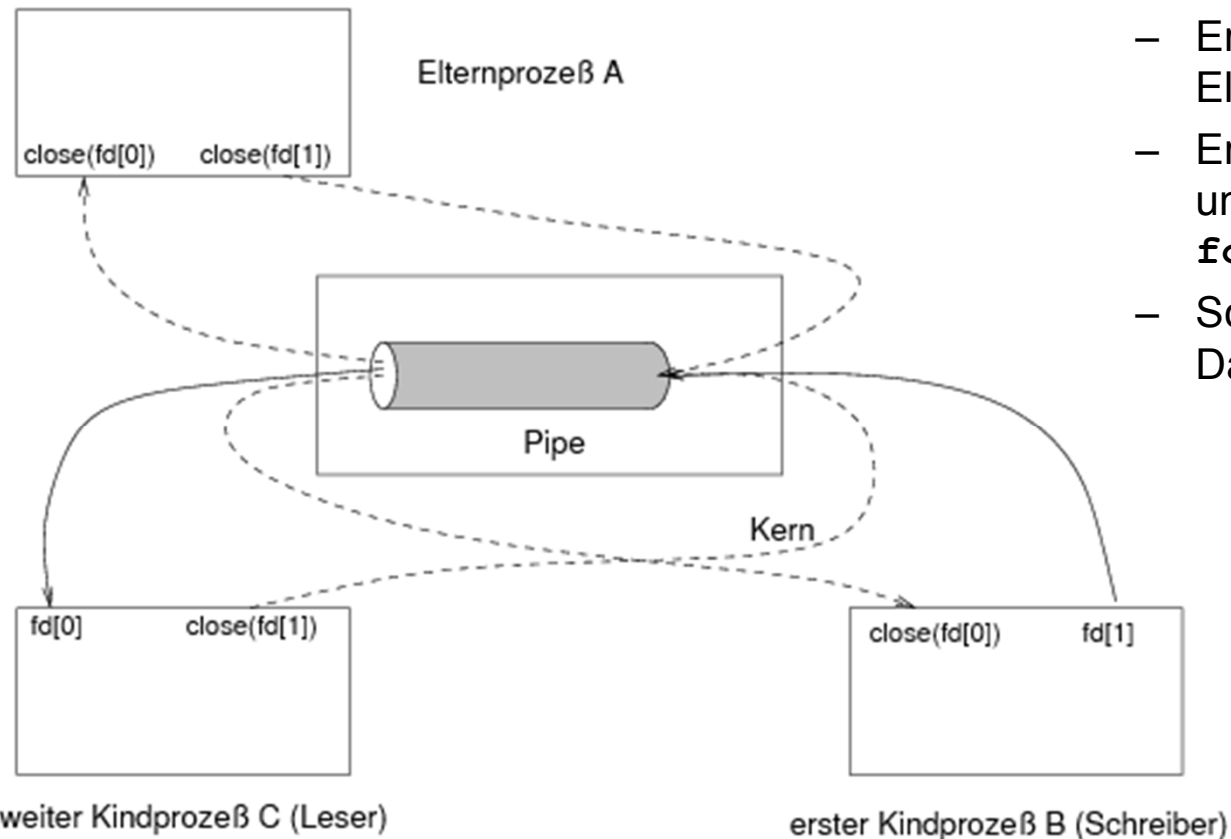
```
/*
 * This program creates a pipe, then forks.
 * The child communicates to the parent over
 * the pipe. Notice that a pipe is a one-way
 * communications device. I can write to the
 * output fd (fds[1], the second file
 * descriptor of the array returned by pipe())
 * and read from the input file descriptor
 * (fds[0]), but not vice versa.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define DATA "Message through the pipe"

main() {
    int fds[2], isParent;
    /* Create a pipe */
    if(pipe(fds) < 0) {
        perror("opening stream fd pair");
        exit(10);
    }...
```

```
...
    if((isParent = fork()) == -1)
        perror("fork");
    else if(isParent) {
        char buf[1024];
        /* This is still the parent.
           It reads the child's message. */
        close(fds[1]);
        if(read(fds[0], buf, sizeof(buf)) < 0)
            perror("reading message");
        printf("-->%s\n", buf);
        close(fds[0]);
    }
    else {
        /* This is the child.
           It writes a message to its parent. */
        close(fds[0]);
        if(write(fds[1], DATA, sizeof(DATA)) < 0)
            perror("writing message");
        close(fds[1]);
    }
}
```

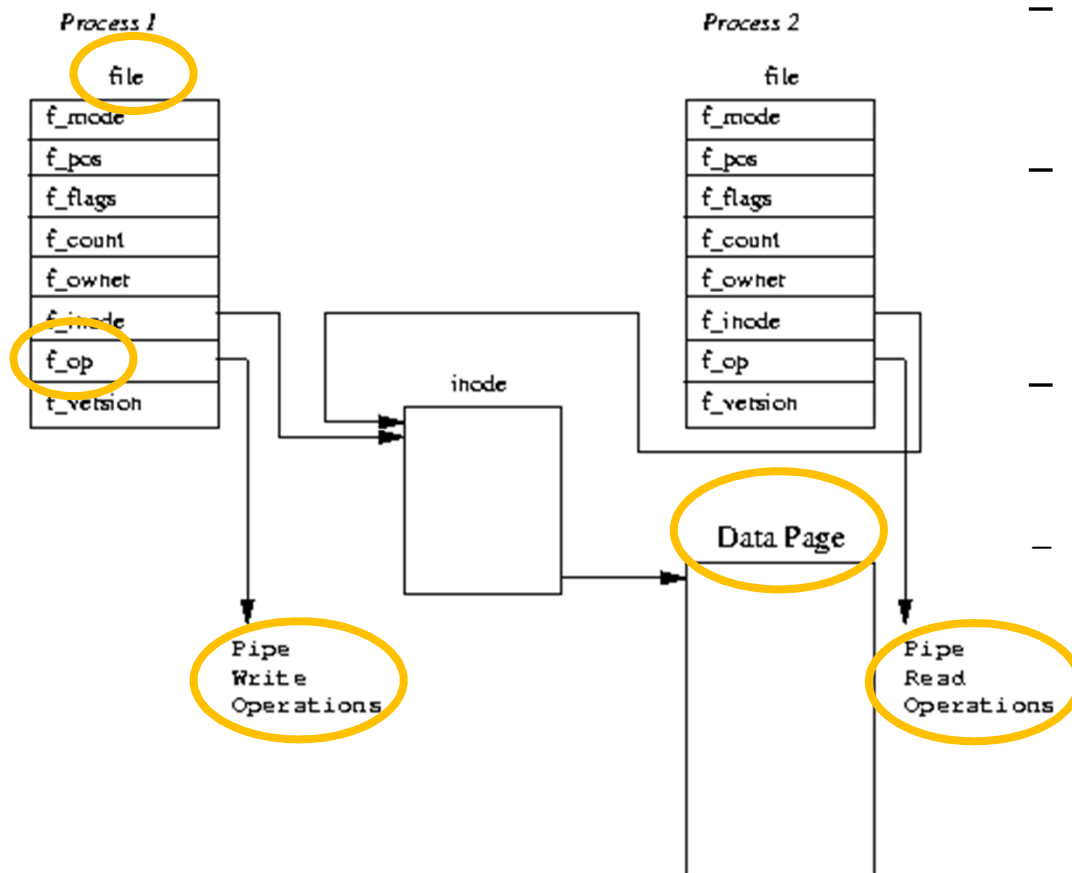
- Beispiel: Kommunikation zwischen zwei Kindprozessen



- Ablauf

- Erzeugen der Pipe durch Elternprozess **A**
- Erzeugen der Kindprozesse **B** und **C** durch zweimaliges `fork` in **A**
- Schließen der Dateideskriptoren
  - `fd[0]` und `fd[1]` in **A**,
  - `fd[0]` in **B**, und
  - `fd[1]` in **C**

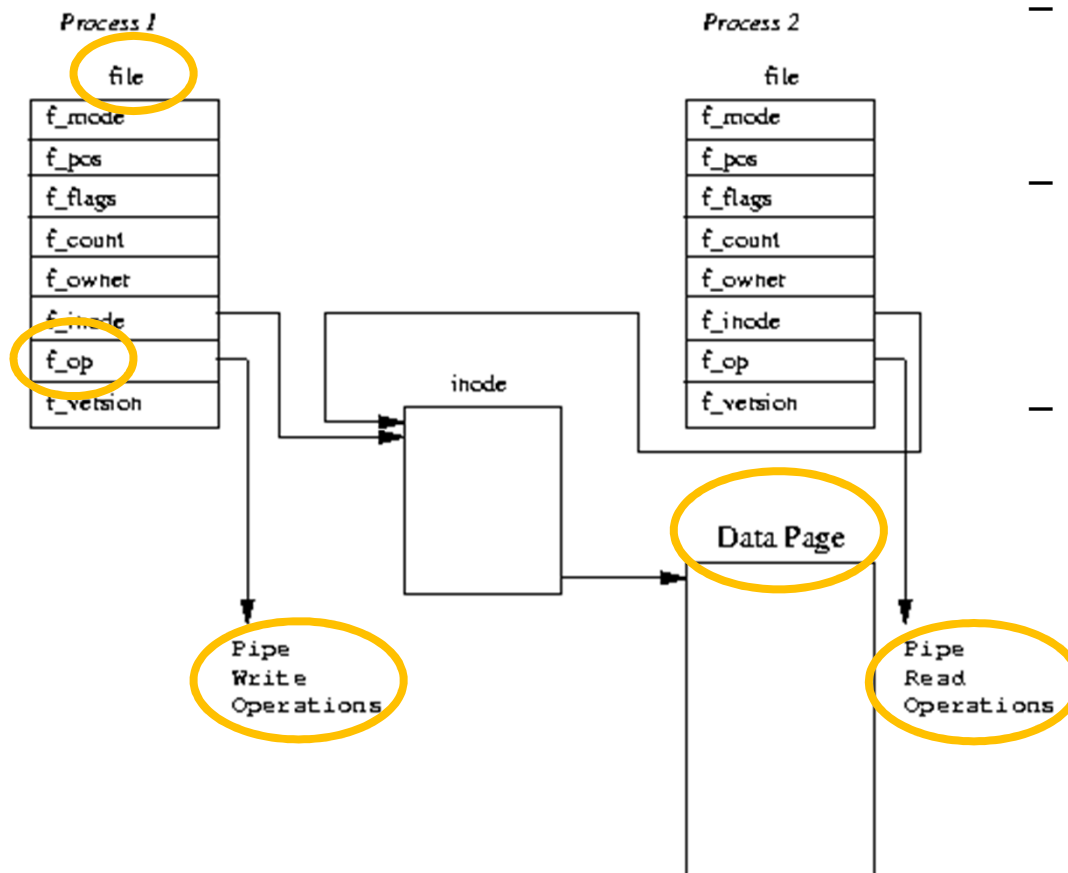
- Beispiel: Linux `pipe()` – Umsetzung (I)
  - Pipe wird über zwei **file** Datenstrukturen (DS) realisiert
  - Datenstrukturen:



- Prozesse 1 und 2 verwalten jeder eine `file` DS für `fd[1]` bzw. `fd[0]`
- Beide `file` DS verweisen auf denselben *Virtual File System (VFS)* Inode (DS, die Datei im Dateisystem repräsentiert)
- Pointer `f_op` verweist auf Lese-/Schreibroutinen für Zugriff auf gemeinsamen Speicher (Data Page)
- `file` DS selbst gibt Lese- bzw. Schreibroutinen vor.

- **Vorteil:** Verbergen des Unterschieds zu Dateizugriffen durch generischen Systemaufrufen.

- Beispiel: Linux `pipe()` – Umsetzung (II)
  - Pipe wird über zwei **file** Datenstrukturen (DS) realisiert
  - Ablauf:



- Prozesse 1 schreibt in Pipe
  - ➡ Daten werden in gem. Speicherbereich kopiert
- Prozess 2 liest von Pipe
  - ➡ Daten werden von gem. Speicherbereich in Adressraum von Prozess 2 kopiert
- **Kernel übernimmt Synchronisation:**
  - Exklusiver Zugriff über Lock
  - Wenn Pipe voll
    - Prozess 1 geht schlafen
    - Signalisierung von Prozess 2 durch Interrupt
  - Ähnliches Verhalten wenn Pipe leer (aber auch non-blocking read erlaubt)



- Named Pipes (FIFO Pipes)
  - Wie (unnamed) Pipe, aber
    - System-persistent
    - zwischen nicht verwandten Prozessen
      - Ansprechbar durch den Namen
    - nicht on-the-fly erstellbar
- Beispiel: Linux
  - Spezielle Dateiart (kein temporäres Objekt!)

```
bash$ mkfifo testfifo
```

```
bash$ ls testfifo -l
```

```
bash$ prw-rw-r- 1 user group 0 Date Time testfifo
```

- Bsp (Nutze Named Pipe um Daten zu komprimieren):

```
bash$ mkfifo testfifo
```

```
bash$ gzip -9 -c < testfifo > out.gz &
```

```
Aufrufender Prozess: bash$ cat file > testfifo
```

- Verwenden dieselben Datenstrukturen wie (unnamed) Pipes
- Kernel kümmert sich, dass lesender Prozess FIFO **vor** schreibendem Prozess öffnet

- Beispiel: Realisierung über Shell-Skripte

```
#!/bin/sh
# receiver.sh
# Open a pipe for reading
FIFO="testfifo"
trap "{ rm -f $FIFO; exit; }" 0 1 2 3 15
mkfifo $FIFO
cat < $FIFO
```

```
#!/bin/sh
# sender.sh
# Open a pipe for writing
FIFO="testfifo"
cat > $FIFO
```

Programm Ablauf

```
bash$ ./receiver.sh
```

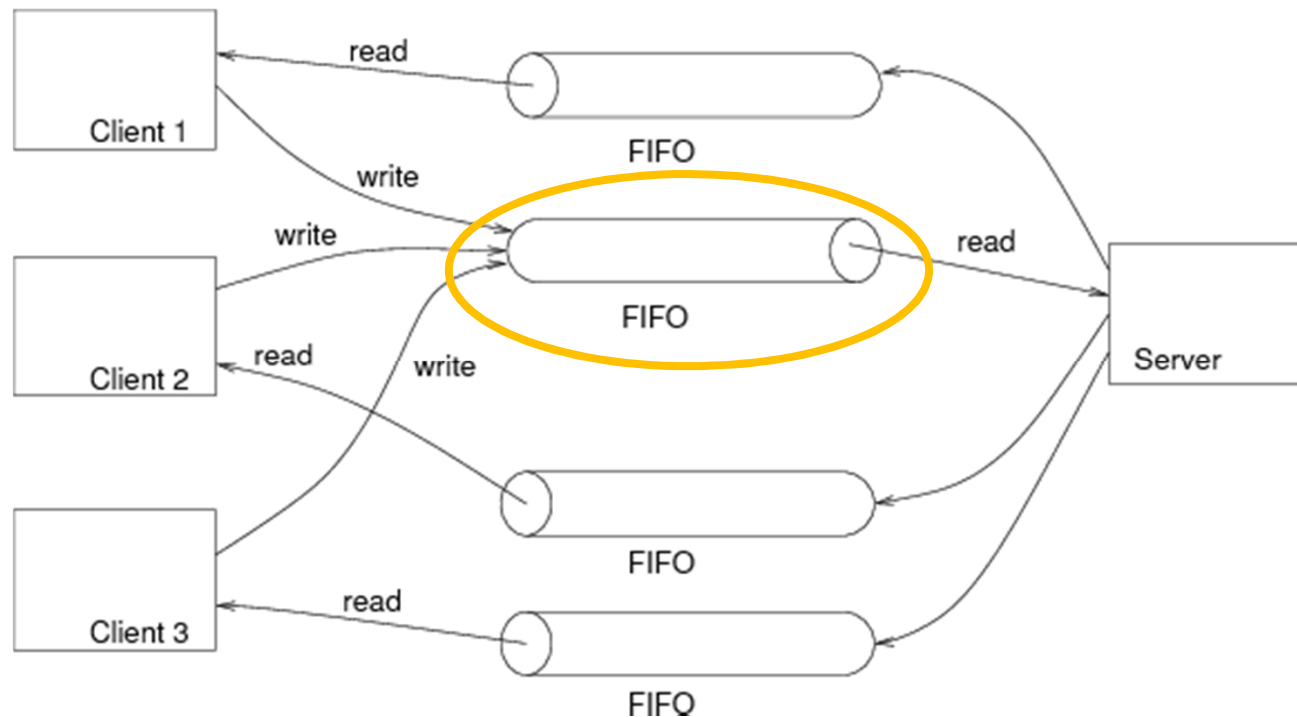
```
bash$ ./sender.sh
hallo receiver
```

```
bash$ ./receiver.sh
hallo server
```

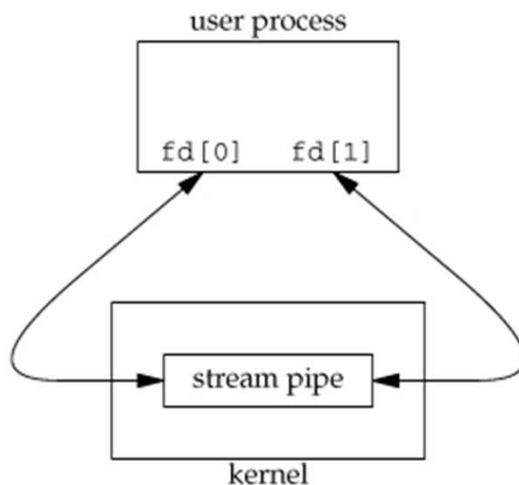
```
bash$ ./sender.sh
hallo receiver
^C
bash$
```

```
bash$ ./receiver.sh
hallo receiver
bash$
```

- Beispiel: Client/Server Kommunikation (I)
  - Server verwendet eine **Input Pipe**
  - Clients verwenden separate Output Pipe
  - Client muss Server Identifikator (z.B. Prozess ID) bereitstellen
  - Client richtet Pipe mit im Pipe-Namen kodierten Identifikator für Server ein



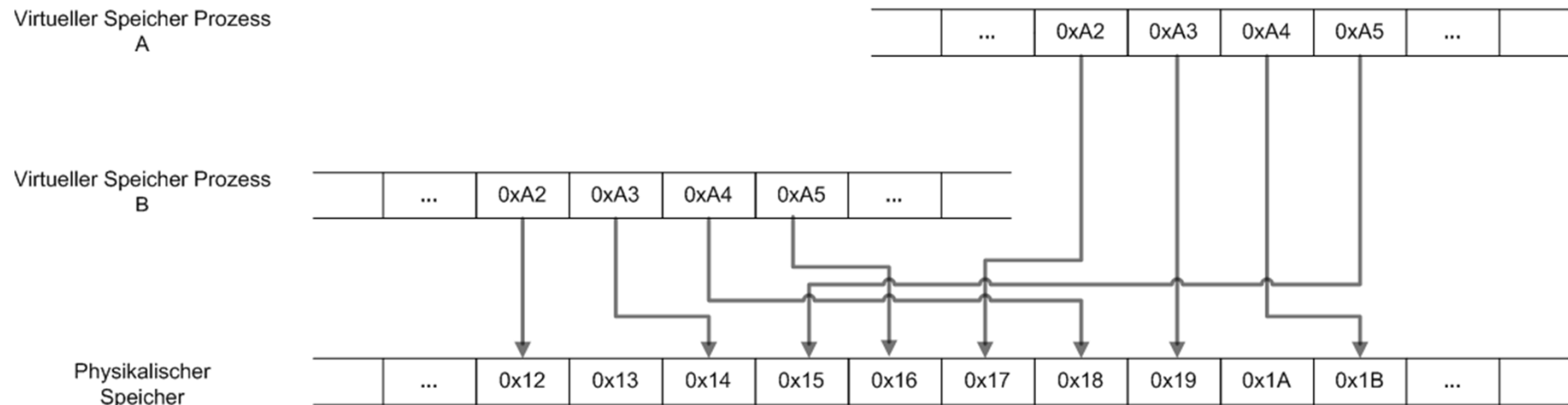
- (Unnamed) Stream Pipes
  - Voll-Duplex => Kommunikation bidirektional
- Beispiel: Unix
  - Funktion `stream_pipe()`
  - Dateideskriptor kann zum Lesen **und** Schreiben verwendet werden
  - Jeder Prozess schließt ein Ende der Pipe und verwendet das andere zum Lesen **und** schreiben.



- Named Stream Pipes
  - Wie (unnamed) Stream Pipe, aber zwischen nicht verwandten Prozessen
  - Problem:
    - Gemeinsamer Schlüssel für Pipe muss beiden Prozessen vorher bekannt sein

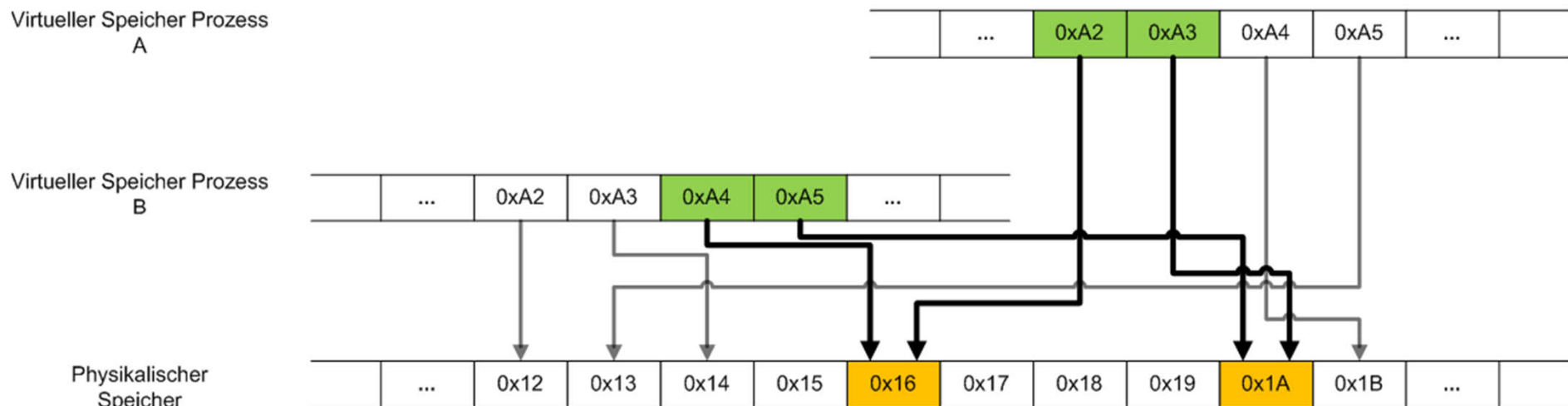
- Virtual Memory
  - Motivation:
    - Physikalischer Speicher zu klein
  - Lösung:
    - Arbeiten mit virtuellem Speicher, der auf physikalischen abgebildet wird
    - Größe des virtuellen Speicher entspricht Größe der maximal adressierbaren Einheit (32 Bit System => Virtueller Speicher umfasst maximal  $2^{32}$  Byte)
  - Umsetzung:
    - Blende virtuelle Seiten die vom rechnenden Prozess gerade benötigt werden in physikalischen Speicher ein
  - Achtung:
    - Jeder Prozess besitzt seinen **eigenen** virtuellen Adressraum!

- Virtual Memory
  - Beispiel



- Mehr zur Adressierung, dem Ein- und Auslagern und der Verwaltung von virtuellem Speicher in einer der kommenden Vorlesungen...

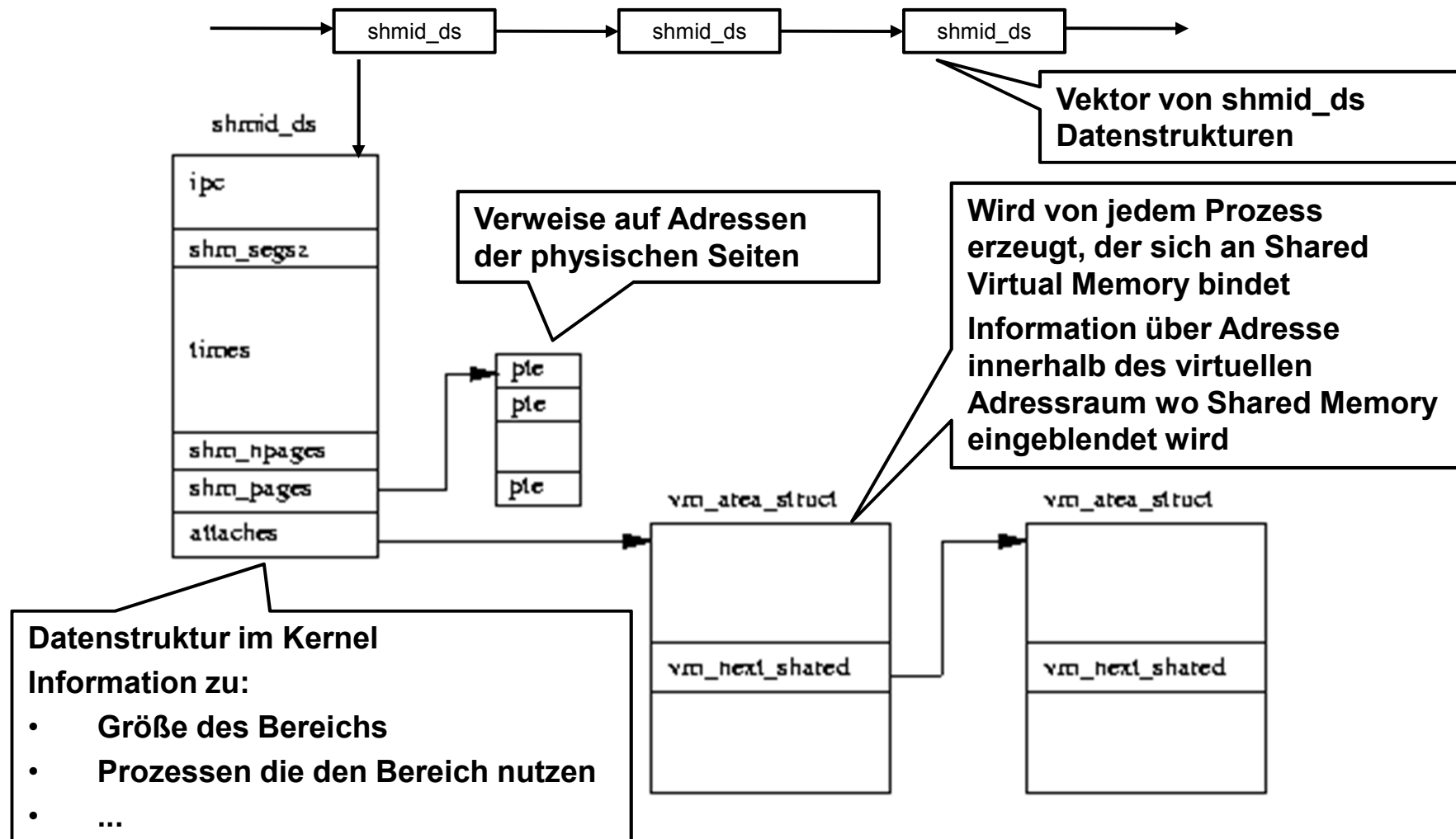
- Shared Virtual Memory
  - Gemeinsame Nutzung von Hauptspeicher durch unabhängige Prozesse
  - Sehr schneller Mechanismus
    - Direkter Zugriff auf Speicher ohne vorhergehendes Kopieren der Daten
  - Einblenden der gemeinsamen Speicherbereichs in den Seitentabellen der kommunizierenden Prozesse
  - Speicherbereich kann für jeden Prozess an verschiedenen Adressen des virtuellen Speichers eingeblendet werden.





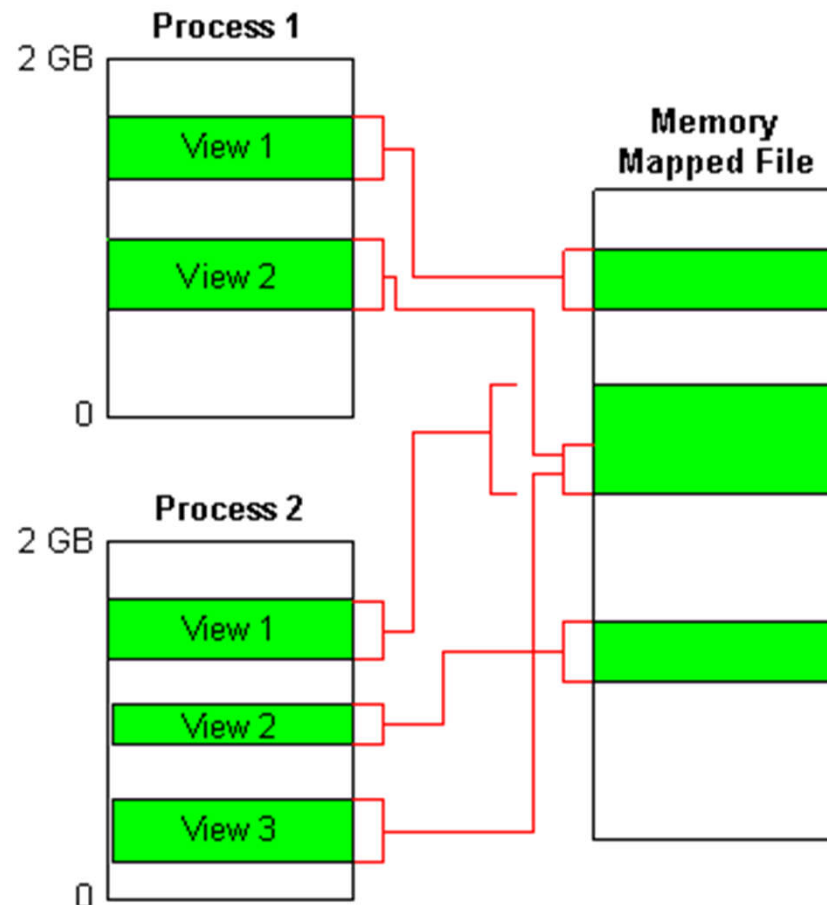
- Probleme:
  - Referenzieren von Shared Virtual Memory über bekannten Schlüssel (vgl. Message Queues)
  - Angabe der Größe schon bei der Erzeugung
  - Synchronisation über anderen IPC-Mechanismus notwendig (Semaphore, Monitor, Datei-Locks, ...)
- Umsetzung
  - Kernel verwaltet Shared Virtual Memory Bereiche in einer Liste
  - Listenelemente verweisen auf Datenstruktur mit Verwaltungsinformationen für Speicherbereich (Zugriffsrechte, ...)
  - Dynamische Allokation von Speicherseiten
    - Wenn Seite noch nicht existiert
      - ⇒ Kernel signalisiert Seitenzugriffsfehler und allokiert Seite

- Beispiel: System V



- Mapped Memory
  - Ähnlich zu Shared Memory
  - Zugriff eines oder mehrerer Prozesse auf Datei des **Hintergrundspeichers** als ob Speicherbereich des Hauptspeichers (Beispiel: Editor)
- Methode:
  - Einblenden bestimmter Datenblöcke in virtuellen Adressbereich eines Prozesses
    - Ermöglicht direkte Modifikation von Dateiinhalten über Zeiger
      - ⇒ Keine bzw. weniger Unterbrechungen da I/O im RAM ausgeführt wird
    - Ermöglicht intelligentes Caching um Systemperformanz zu erhöhen
      - Synchronisation modifizierter Daten im Hauptspeicher mit Hintergrundspeicher ist Aufgabe des Betriebssystems
- Problem:
  - Mögliche Performanz Einbußen bei parallelem Zugriff
    - Mehrere Prozesse mit Schreibrechten auf Mapped Memory Segment
      - ⇒ Höherer Synchronisationsaufwand

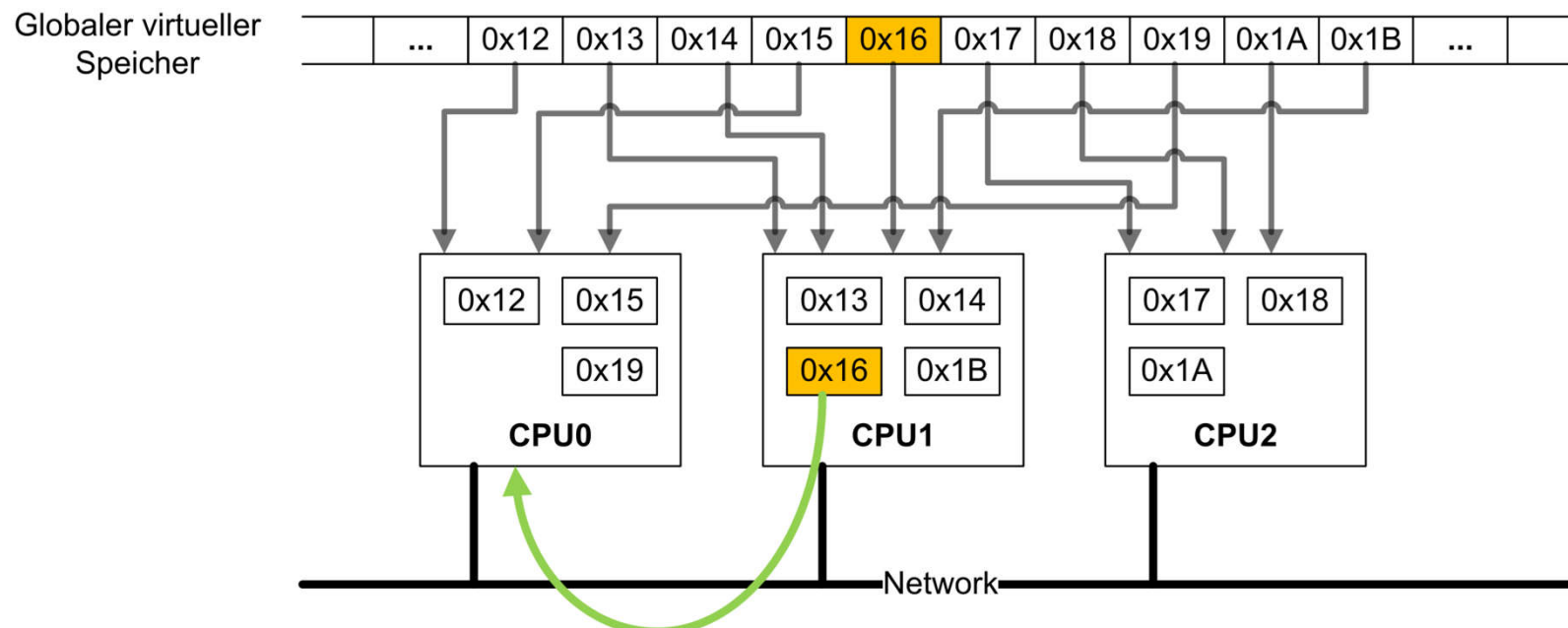
- Beispiel: Windows NT



- Realisierung über Views
  - Einblenden bestimmter Speicherbereiche als View
  - Bereiche können mehrfach eingebunden werden
  - Bereiche können sich überlappen
  - Problem:
    - benötigt geeignete Synchronisation
- Unter Linux Memory Mapping z.B. mit `mmap()` und `munmap()`

- Distributed Shared Memory (nur kurz skizziert)
  - Ähnlich zu Shared Virtual Memory, aber über mehrere Systeme hinweg
  - Jedes System hat seinen eigenen virtuellen Speicher und eigene Seitentabellen
  - Wenn CPU Seite lesen / schreiben will, die nicht lokal vorhanden ist
    - ⇒ Trap
      - Generiere Page Fault
      - Suche Seite
      - Hole Seite
- Umsetzung:
  - I.d.R. eigener Middleware Ansatz
- Problem: Performanzeinbußen
  - Schutz vor parallelem Zugriff (Locks)
  - Keine Kontrolle durch Programmierer über generierte Nachrichten

- Beispiel:
  - CPU0 will auf Seite 0x16 zugreifen
  - Seite lokal nicht verfügbar
  - ➡ Trap
    - Page Fault
    - Seite suchen
    - Seite holen



- Message Passing
  - **Direkte** Kommunikation zwischen Prozessen, die auf **verschiedenen CPUs eines Systems** ausgeführt werden
  - Prozesse können sich gegenseitig Nachrichten schicken
    - Für Prozesse auf verschiedenen Systemen
      - ⇒ Andere Techniken (RPC,...)
- Umsetzung
  - Zwei Prozeduren (siehe abstrakte Sichtweise)
    - `send(Zieladresse, Nachricht)`
    - `receive(Quelleadresse, Nachricht)`
      - ⇒ Quelleadresse notwendig falls Zielprozess Nachrichtenempfang auf bestimmte Prozesse einschränken will

- Synchronisation:
  - Blocking Send, Blocking Receive
    - Sender und Empfänger werden blockiert, bis Nachricht ausgeliefert wurde
    - Ermöglicht feste Synchronisation zw. 2 Prozessen => Rendezvous
  - Nonblocking Send, Blocking Receive
    - Sender kann nach dem Abschicken einer Nachricht weiterarbeiten
    - Empfänger ist bis zum Erhalt der Nachricht blockiert
    - Sinnvollste Kombination: Erlaubt das schnelle Verschicken verschiedener Nachrichten an mehrere Empfänger
    - Bsp.: Server-Prozess, der Dienste bereitstellt
  - Nonblocking Send, Nonblocking Receive
    - Weder Sender noch Empfänger müssen warten  
⇒ Nachrichten können verloren gehen!

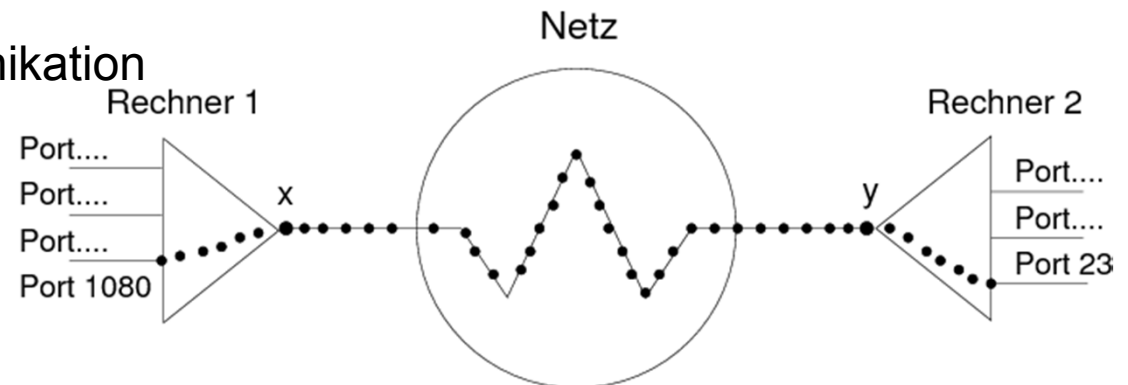


- Verwendung eines Empfangs- u. Sende-Puffers möglich
  - Nachrichten werden gepuffert und später gesendet bzw. empfangen
  - Asynchrone und Synchrone Vorgehensweise
- Asynchrones (nicht-blockierendes) Message Passing (I)
  - Problem beim Senden:
    - Kernel muss sicherstellen, dass Sender nicht in Puffer der Nachricht schreibt, solange Kernel das Senden noch nicht abgeschlossen hat
  - Lösungen:
    - Kopieren des Puffers in Kernel Adressraum
      - Performanz Probleme
    - User Level Interrupt, sobald Kernel fertig mit Sendevorgang
      - Schwieriges Debugging (Race Conditions), da schlecht reproduzierbar
    - Copy on Write: Puffer nur kopieren, wenn Prozess versucht in Sendepuffer zu schreiben (vgl. `fork()`)

- Asynchrones (nicht-blockierendes) Message Passing (II)
  - Problem beim Empfangen:
    - Kernel darf nicht in den Puffer der Nachricht schreiben, solange Empfänger das Lesen noch nicht abgeschlossen hat
  - Lösungen (I):
    - `receive()` bewirkt im Kernel Bekanntgabe der Adresse des Empfangspuffers  
User Level Interrupt, sobald Kernel fertig mit Empfangsvorgang  
⇒ User Level Interrupts anfällig in der Programmierung
    - Polling:
      - Aufruf von `poll()` in einer Schleife
      - Nachricht vorhanden ⇒ Aufruf kehrt sofort zurück
      - Sonst ⇒ Rückkehr sobald Nachricht eintrifft
    - ⇒ Sehr effizient (kein Threading notwendig), aber fehleranfällig in der Programmierung
  - Pop-Up Thread: Neuer Thread wenn Nachricht eintrifft  
⇒ Großer Overhead bei vielen kleinen Nachrichten

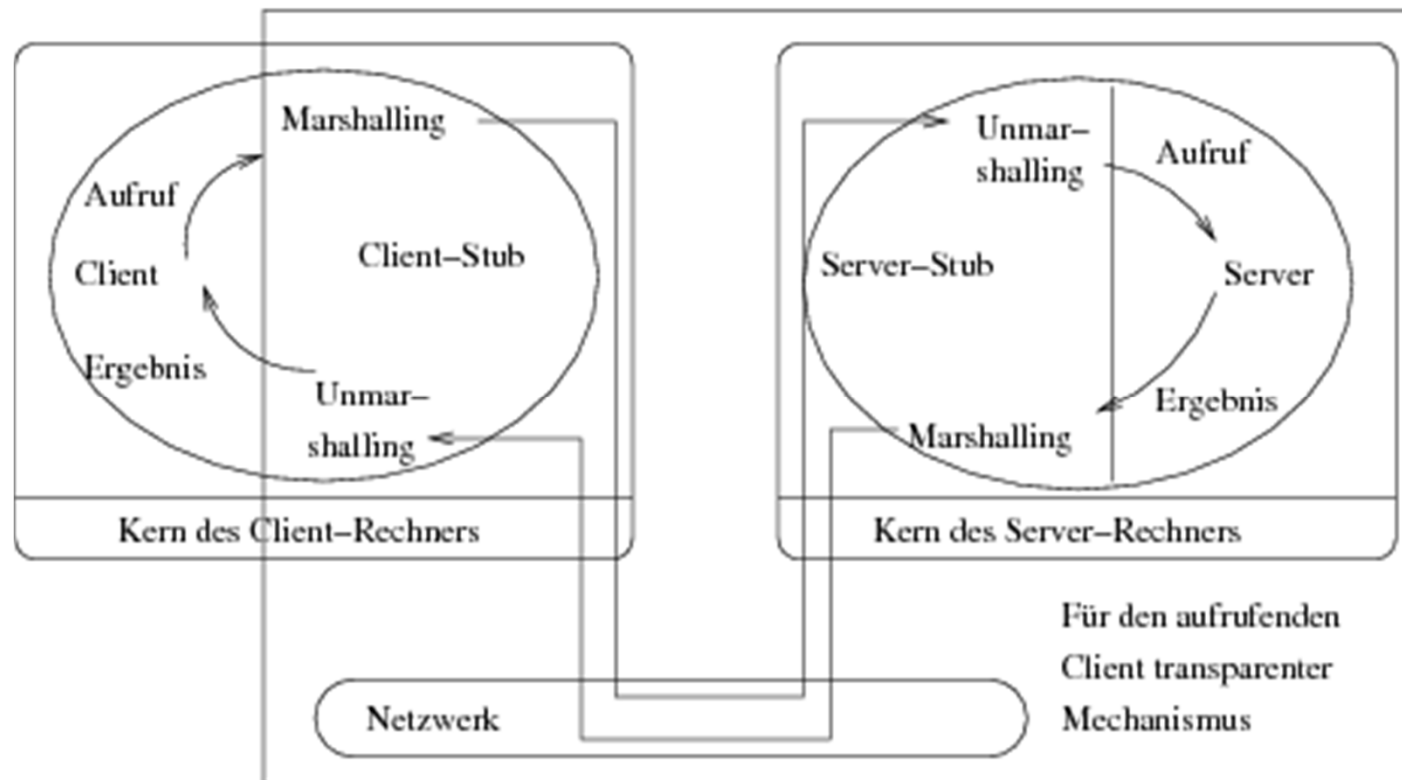
- Synchrones (blockierendes) Message Passing
  - Redundanz
  - Problem beim Senden:
    - CPU ist **idle** solange Sendepuffer nicht geleert
    - Lösungen:
      - Eigener Thread für Sendevorgang
  - Problem beim Empfangen:
    - Empfänger blockiert solange keine Nachricht ankommt
    - Lösung:
      - Eigener Thread für Empfangsvorgang

- Sockets
  - Dienen als wohldefinierter Kommunikationsendpunkt
  - Kommunikation zwischen unabhängigen lokalen und entfernten Prozessen
  - Ermöglichen
    - Zentrale Verbindungsverwaltung (Ports)
    - Zuweisung zu Prozessen
    - Identifikation von Diensten (Port 22 => SSH)
- Unterscheidung
  - Verbindungsorientierte Kommunikation
    - Datenströme/Stream Socket
    - TCP
  - Verbindungslose Kommunikation
    - Datagramme
    - UDP



- RPC (Remote Procedure Call)
  - Idee:
    - Führe Interaktion zwischen Prozessen, die auf verschiedenen Rechnern laufen und miteinander kommunizieren müssen, auf einen entfernten Prozeduraufruf zurück, der für die beteiligten Prozesse wie ein lokaler Prozedur Aufruf erscheint.
- Umsetzung:
  - Spezielle Stub-Objekte kümmern sich um
    - Verpacken (Marshalling)
    - Senden
    - Empfangen, und
    - Entpacken der Funktionsparameter (Unmarshalling)
  - Client Prozess benötigt Client-Stub Bibliothek
  - Server Prozess benötigt Server-Stub Bibliothek.

- Beispiel



- Ablauf (I):
  - Client
    - Ruft Client-Stub-Methode auf
      - ⇒ dient als Proxy für entfernte Prozedur
    - Client-Stub verpackt Funktionsparameter in eine Nachricht (Marshalling)
    - Client-Stub beauftragt Kern durch das send-Primitiv Nachricht an Server zu schicken
    - Client-Stub ruft **receive()** auf
      - ⇒ Blockierung, bis Antwort eintritt.
  - Server:
    - Kernel empfängt Nachrichten und übergibt diese an Server-Stub (wartet im **receive()**)
    - Server-Stub entpackt Parameter aus Nachricht (Unmarshalling) und ruft entsprechende Prozedur lokal auf.
    - Nach Ausführung des Unterprogramms
      - ⇒ Rückgabe der Ergebnisse an Server-Stub

- Ablauf (II):
    - Server:
      - Server-Stub verpackt Ergebnis (Marshalling) und übergibt sie an **send()** - Direktive des Kern
      - Server-Stub ruft wieder **receive()** auf
    - Client
      - Kernel empfängt Nachricht und übergibt sie an Client-Stub → Blockierung auf Clientseite beendet
      - Nachricht wird entpackt (Unmarshalling)
      - Ergebnis wird in entsprechenden Adressbereich bzw. Registern abgelegt
      - Rückkehr vom Prozeduraufruf
- ⇒ Client hat nichts von entferntem Aufruf mitbekommen.



- Probleme (hier nur ein paar):
  - Un-/Marshalling benötigt Standardrepräsentation
    - Big-/Little Endian
    - Parameters
  - Call-by-reference schwierig
    - Client und Server haben keinen gemeinsamen Speicherbereich
    - Mögliche Lösung: Copy-Restore
- Bessere Ansätze:
  - Remote Methode Invocation (RMI)
    - Vordefinierte Interfaces (CORBA IDL)
  - SOAP / Webservices, XML-RPC, JSON-RPC, ...

## Interprozesskommunikation (IPC) zur Kommunikation:

- Ziel: Prozesse wollen untereinander kommunizieren
- Methoden:
  - Message Queues
  - Pipes
    - Unnamed/Named
    - Unnamed/Named Stream Pipes
  - Memory
    - Shared Virtual Memory
    - Mapped Memory
    - Distributed Shared Memory
  - Message Passing
  - Sockets
  - RPC