



TECHNISCHE  
UNIVERSITÄT  
DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

# BETRIEBSSYSTEME UND SICHERHEIT

mit Material von Olaf Spinczyk,  
Universität Osnabrück

*Interprozesskommunikation*

<https://tud.de/inf/os/studium/vorlesungen/bs>

**HORST SCHIRMEIER**

# Inhalt

- Wiederholung
- Grundlagen der Interprozesskommunikation
- Lokale Interprozesskommunikation unter UNIX
  - Signale
  - *Pipes*
  - *Message Queues*
- Rechnerübergreifende Interprozesskommunikation
  - *Sockets*
  - Entfernte Prozeduraufrufe (RPCs)
- Zusammenfassung

Silberschatz, Kap. ...

3.4: *Interprocess Communication*

Tanenbaum, Kap. ...

2.3: *Interprocess Communication*  
(leider nur sehr wenig dazu!)

# Inhalt

- **Wiederholung**
- Grundlagen der Interprozesskommunikation
- Lokale Interprozesskommunikation unter UNIX
  - Signale
  - *Pipes*
  - *Message Queues*
- Rechnerübergreifende Interprozesskommunikation
  - *Sockets*
  - Entfernte Prozeduraufrufe (RPCs)
- Zusammenfassung

# Wiederholung

- Prozesse können miteinander interagieren
  - Aufeinander warten (**Synchronisation**)
  - Daten austauschen (**Kommunikation**)
- Wartemechanismen ...
  - sind notwendig für kontrollierte Kommunikation
  - können zu Verklemmungen führen
- Datenaustausch wurde bisher nur am Rande betrachtet
  - Leicht- und federgewichtige Prozesse im selben Adressraum

# Inhalt

- Wiederholung
- **Grundlagen der Interprozesskommunikation**
- Lokale Interprozesskommunikation unter UNIX
  - Signale
  - *Pipes*
  - *Message Queues*
- Rechnerübergreifende Interprozesskommunikation
  - *Sockets*
  - Entfernte Prozeduraufrufe (RPCs)
- Zusammenfassung

# Interprozesskommunikation

## *Inter-Process Communication (IPC)*

- Mehrere Prozesse bearbeiten **eine** Aufgabe:
  - **gleichzeitige Nutzung** von zur Verfügung stehender Information durch mehrere Prozesse
  - **Verkürzung der Bearbeitungszeit** durch Parallelisierung
  - **Verbergen von Bearbeitungszeiten** durch Ausführung „im Hintergrund“
- Kommunikation durch gemeinsamen Speicher
  - Datenaustausch nebenläufiges Schreiben in bzw. Lesen aus einem gemeinsamen Speicher,
  - dabei muss auf Synchronisation geachtet werden.
- **Heute: Kommunikation durch Nachrichten**
  - Nachrichten werden zwischen Prozessen ausgetauscht,
  - gemeinsamer Speicher ist nicht erforderlich.

# Nachrichtenbasierte Kommunikation

- ... basiert auf zwei Primitiven:

**send** (*Ziel, Nachricht*)

**receive** (*Quelle, Nachricht*)

(oder ähnlich)

- Unterschiede gibt es in ...
  - Synchronisation
  - Adressierung
  - und diversen anderen Eigenschaften

# Synchronisation

... bei nachrichtenbasierter Kommunikation

- **Synchronisation bei Senden / Empfangen**
  - **Synchroner Nachrichtenaustausch** (auch „Rendezvous“)
    - Empfänger blockiert bis die Nachricht eingetroffen ist.
    - Sender blockiert bis die Ankunft der Nachricht bestätigt ist.
  - **Asynchroner Nachrichtenaustausch**
    - Sender gibt die Nachricht dem Betriebssystem und arbeitet weiter
    - Blockierung auf beiden Seiten optional
    - Pufferung immer erforderlich
- Häufig anzutreffen:
  - **Asynchroner** Nachrichtenaustausch  
mit **potenziell blockierendem** Senden und Empfangen



# Adressierung

... bei nachrichtenbasierter Kommunikation

- **Direkte Adressierung**
  - Prozess-ID (Signale)
  - Kommunikationsendpunkt eines Prozesses (*Port, Socket*)
- **Indirekte Adressierung**
  - Kanäle (*Pipes*)
  - Briefkästen (*Mailboxes*), Nachrichtenpuffer (*Message Queues*)
- Zusätzliche Dimension: **Gruppenadressierung**
  - **Unicast** – an genau einen
  - **Multicast** – an eine Auswahl
  - **Broadcast** – an alle

# Diverse andere Eigenschaften

... bei nachrichtenbasierter Kommunikation

- **Nachrichtenformat**

- Stromorientiert / nachrichtenorientiert
- Feste Länge / variable Länge
- Getypt / ungetypt

- **Übertragung**

- Unidirektional / Bidirektional (halb-duplex, voll-duplex)
- zuverlässig / unzuverlässig
- Reihenfolge bleibt erhalten / nicht erhalten

# Inhalt

- Wiederholung
- Grundlagen der Interprozesskommunikation
- **Lokale Interprozesskommunikation unter UNIX**
  - Signale
  - *Pipes*
  - *Message Queues*
- Rechnerübergreifende Interprozesskommunikation
  - *Sockets*
  - Entfernte Prozeduraufrufe (RPCs)
- Zusammenfassung

# UNIX-Signale

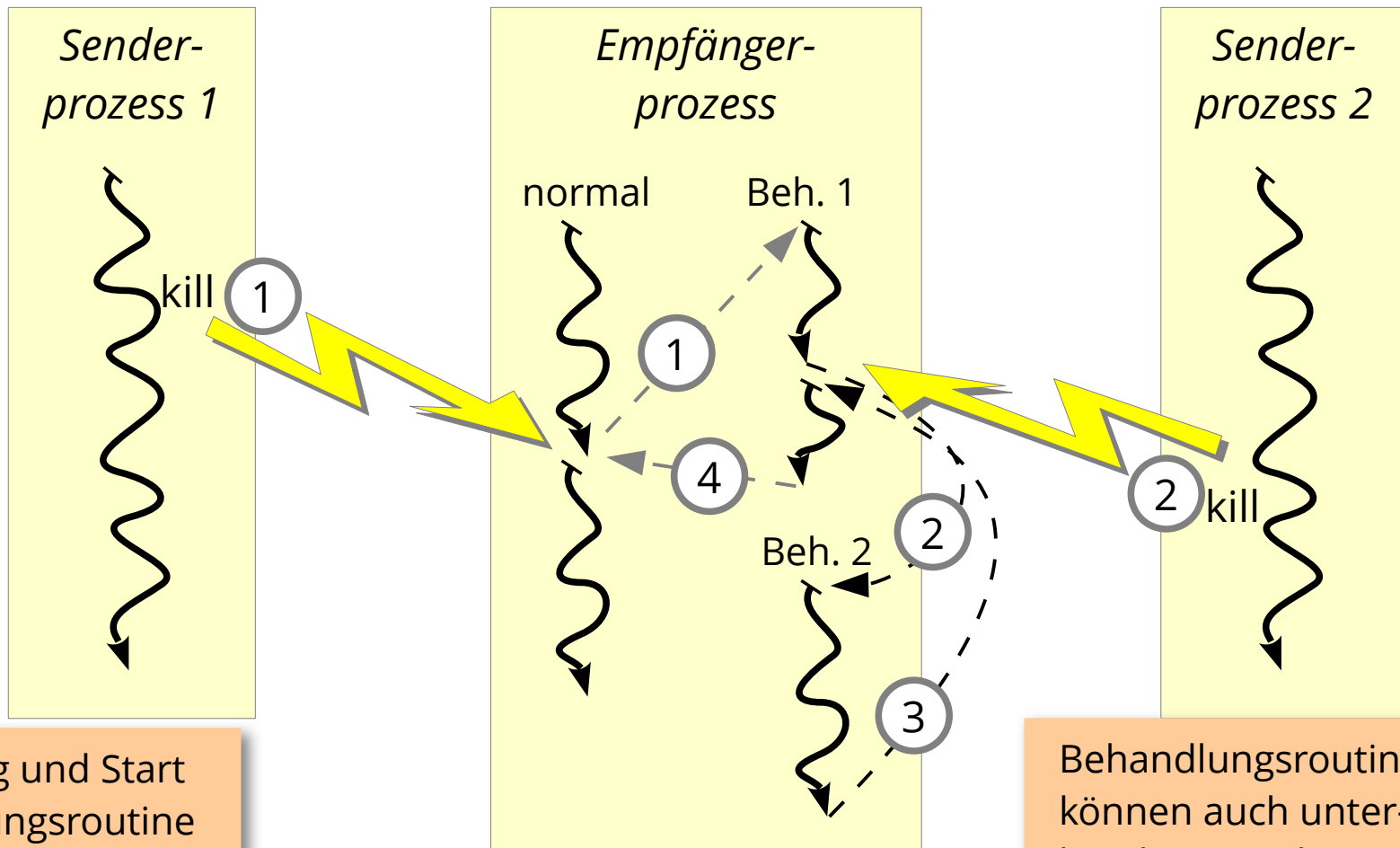
- Signale sind in Software **nachgebildete Unterbrechungen**
  - ähnlich denen eines Prozessors durch E/A-Geräte
  - minimale Form der Interprozesskommunikation (Übertragung der Signalnummer)
- Sender:
  - **Prozesse:** mit Hilfe des Systemaufrufs **kill** (2)
  - **Betriebssystem:** bei Auftreten bestimmter Ereignisse
- Empfänger-Prozess führt Signalbehandlung durch:
  - Ignorieren,
  - Terminierung des Prozesses oder
  - Aufruf einer Behandlungsfunktion
    - Nach der Behandlung läuft Prozess an unterbrochener Stelle weiter.

# Signale

- Mit Hilfe von Signalen können Prozesse über Ausnahmesituation informiert werden
  - ähnlich wie Hardwareunterbrechungen
- Beispiele:
  - **SIGINT** Prozess abbrechen (z.B. bei Ctrl-C)
  - **SIGSTOP** Prozess anhalten (z.B. bei Ctrl-Z)
  - **SIGWINCH** Fenstergröße wurde geändert
  - **SIGCHLD** Kindprozess terminiert
  - **SIGSEGV** Speicherschutzverletzung des Prozesses
  - **SIGKILL** Prozess wird getötet
  - ...
- Die Standardbehandlung (terminieren, anhalten, ...) kann für die meisten Signale überdefiniert werden.
  - siehe **sigaction** (2) (klassisch: **signal** (2))

# UNIX-Signale: Logische Sicht

Hollywood-Prinzip: *"Don't call us, we'll call you."*



Signalisierung und Start der Behandlungsroutine erfolgen in der logischen Sicht gleichzeitig.

Behandlungsroutinen können auch unterbrochen werden.

# UNIX-Signale: Technische Sicht

- Die Signalbehandlung erfolgt immer beim Übergang vom *Kernel* in der *User Mode*.
- Was passiert also wirklich, wenn der Zielprozess gerade ...
  - 1) läuft, also im Zustand **RUNNING** ist (z.B. *Segmentation Fault*, *Bus Error*)?
    - **Unmittelbarer Start** der Behandlungsroutine
  - 2) gerade nicht läuft, aber **READY** ist (z.B. Systemaufruf **kill**)?
    - Im **Prozesskontrollblock** wird das Signal vermerkt.
    - Wenn der Prozess die CPU zugeteilt bekommt, erfolgt die Behandlung.
  - 3) auf E/A wartet, also **BLOCKED** ist?
    - Der E/A-Systemaufruf (z.B. **read**) wird mit Fehlercode EINTR **abgebrochen**.
    - Der Prozesszustand wird auf **READY** gesetzt.
    - Danach wie bei 2)
    - Ggf. wird der unterbrochene Systemaufruf neu ausgeführt (SA\_RESTART).

# UNIX-Signale: Beispiel

Auszug aus dem Handbuch  
des *Apache* HTTP-Servers

## Stopping and Restarting Apache

To send a signal to the parent you should issue a command such as:

```
kill -TERM `cat /usr/local/apache/logs/httpd.pid`
```

### **TERM Signal: stop now**

Sending the TERM signal to the parent causes it to immediately attempt to **kill off all of its children**. It may take it several seconds to complete killing off its children. Then the parent itself exits. Any requests in progress are terminated, and no further requests are served.

### **HUP Signal: restart now**

Sending the HUP signal to the parent causes it to kill off its children like in TERM but the parent doesn't exit. It **re-reads its configuration files**, and **re-opens any log files**. Then it spawns a new set of children and continues serving hits.

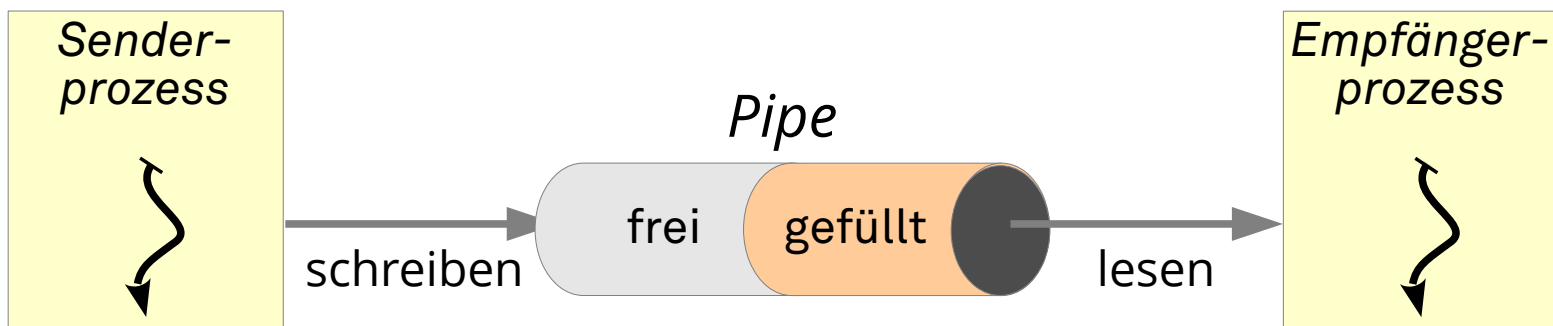
### **USR1 Signal: graceful restart**

The USR1 signal causes the parent process to advise the children to exit after their current request (or to exit immediately if they're not serving anything). The parent re-reads its configuration files and re-opens its log files. As each child dies off the parent replaces it with a child from the new generation of the configuration, which begins serving new requests immediately.



# UNIX Pipes

- Kanal zwischen zwei Kommunikationspartnern
  - unidirektional
  - gepuffert (feste Puffergröße)
  - zuverlässig
  - stromorientiert
- Operationen: Schreiben und Lesen
  - Ordnung der Zeichen bleibt erhalten (Zeichenstrom)
  - Blockierung bei voller *Pipe* (Schreiben) und leerer *Pipe* (Lesen)



# UNIX *Pipes*: Programmierung

- **Unbenannte *Pipes***

- Erzeugen einer Pipe: `int pipe(int fdes[2])`
- Nach erfolgreichem Aufruf (Rückgabewert == 0) kann man ...
  - über `fdes[0]` aus der *Pipe* lesen (Systemaufruf **read**)
  - über `fdes[1]` in die *Pipe* schreiben (Systemaufruf **write**)
- Nun muss man nur noch das eine Ende an einen anderen Prozess weitergeben.  
(siehe nächste Folie)

- **Benannte *Pipes***

- *Pipes* können auch als Spezialdateien ins Dateisystem gelegt werden:  
`int mkfifo(<Dateiname>, mode_t mode)`
- Standardfunktionen zum Öffnen, Lesen, Schreiben und Schließen können dann verwendet werden.
  - Normale Dateizugriffsrechte regeln, wer die *Pipe* benutzen darf.

# UNIX Pipes: Programmierung

```
enum { READ=0, WRITE=1 };

int main (int argc, char *argv[]) {
    int res, fd[2];
    if (pipe (fd) == 0) {                /* Pipe erzeugen */
        res = fork();
        if (res > 0) {                  /* Elternprozess */
            close(fd[READ]);             /* Leseseite schließen */
            dup2(fd[WRITE], 1);          /* Std-Ausgabe in Pipe */
            close(fd[WRITE]);            /* Deskriptor freigeben */
            execlp(argv[1], argv[1], NULL); /* Schreiber ausführen */
        }
        else if (res == 0) {            /* Kindprozess */
            close(fd[WRITE]);            /* Schreibseite schließen */
            dup2(fd[READ], 0);           /* Std-Eingabe aus Pipe */
            close(fd[READ]);             /* Deskriptor freigeben */
            execlp(argv[2], argv[2], NULL); /* Leser ausführen */
        }
    }
    ...Fehler behandeln
}
```

# UNIX Pipes: Programmierung

```
enum { READ=0, WRITE=1 };
```

```
int main (int argc, char *argv[]) {
```

```
    int res, fd[2];
```

```
    if (pipe (fd) == 0) {
```

```
        res = fork();
```

```
        if (res > 0) {
```

```
            /* Pipe erzeugt */
```

```
            /* Elternprozess */
```

„./connect ls wc“ entspricht dem Shell-Kommando „ls | wc“

```
olaf@xantos:~/V_BS/vorlesung/code> ls
```

```
connect connect.c execl.c fork.c orphan.c wait.c
```

```
olaf@xantos:~/V_BS/vorlesung/code> ./connect ls wc
```

```
        6        6        49
```

```
        close (fd[WRITE]);
```

```
        dup2 (fd[READ], 0);
```

```
        close (fd[READ]);
```

```
        execlp (argv[2], argv[2], NULL);
```

```
            /* Schreibseite schließen */
```

```
            /* Std-Eingabe aus Pipe */
```

```
            /* Deskriptor freigeben */
```

```
            /* Leser ausführen */
```

```
    }
```

```
}
```

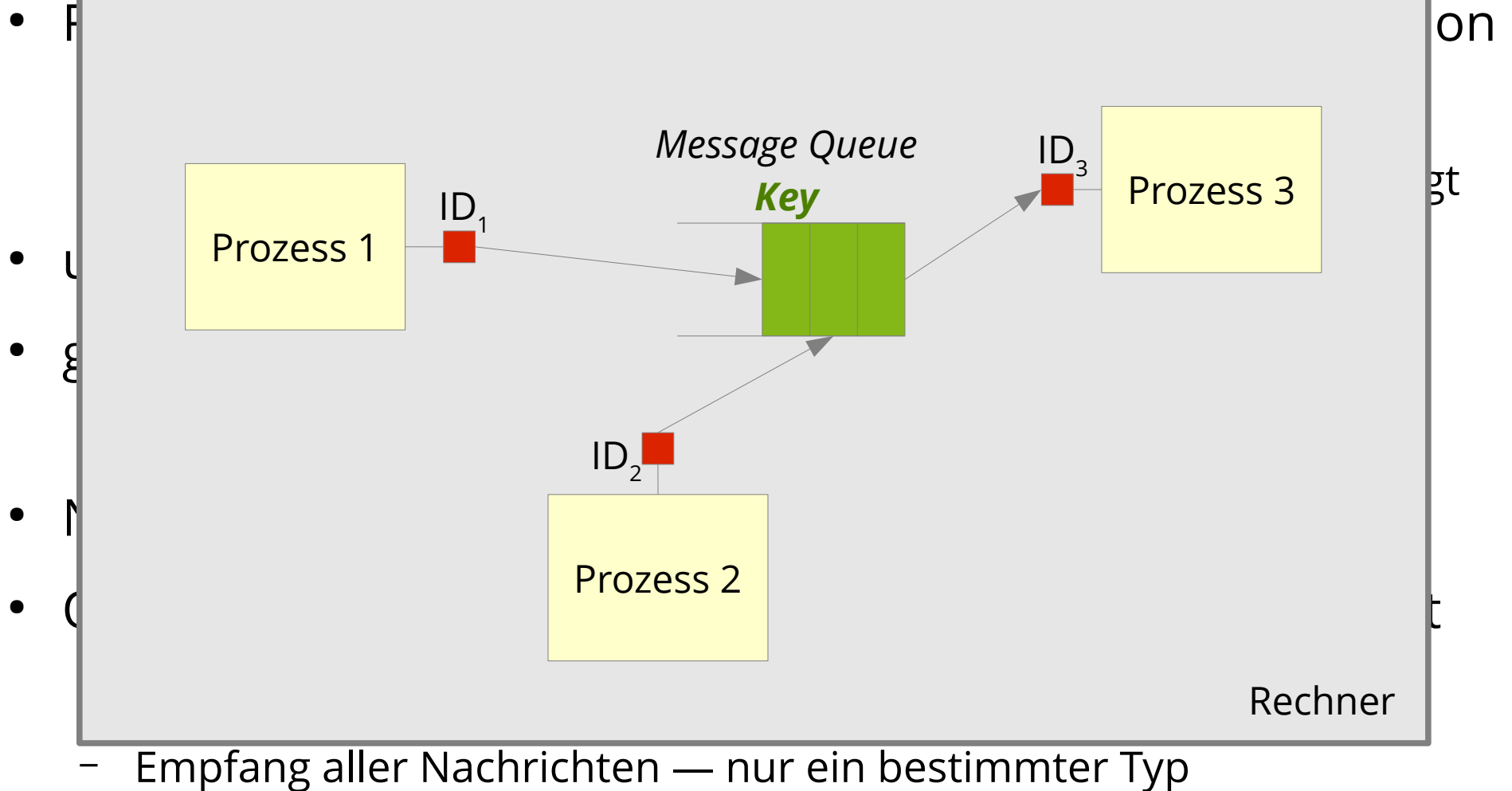
```
...Fehler behandeln
```

```
}
```

# UNIX *Message Queues*

- Rechnerweit eindeutige Adresse (*Key*) dient zur Identifikation
  - Zugriffsrechte wie auf Dateien
  - Prozesslokale Nummer (*MsqID*) wird bei allen Operationen benötigt
- ungerichtete M:N-Kommunikation
- gepuffert
  - einstellbare Größe pro *Queue*
- Nachrichten haben einen Typ (long-Wert)
- Operationen zum Senden und Empfangen einer Nachricht
  - blockierend — nicht-blockierend (aber nicht asynchron)
  - Empfang aller Nachrichten — nur ein bestimmter Typ

# UNIX Message Queues



# UNIX *Message Queues*: Programmierung / Beispiel

[intentionally left blank]

*Message Queues* werden heutzutage nur noch selten eingesetzt, da sie anders als Sockets (siehe nächster Abschnitt) **auf lokale Kommunikation beschränkt** sind. Zudem ist der Anwendungscode **weniger portabel**.

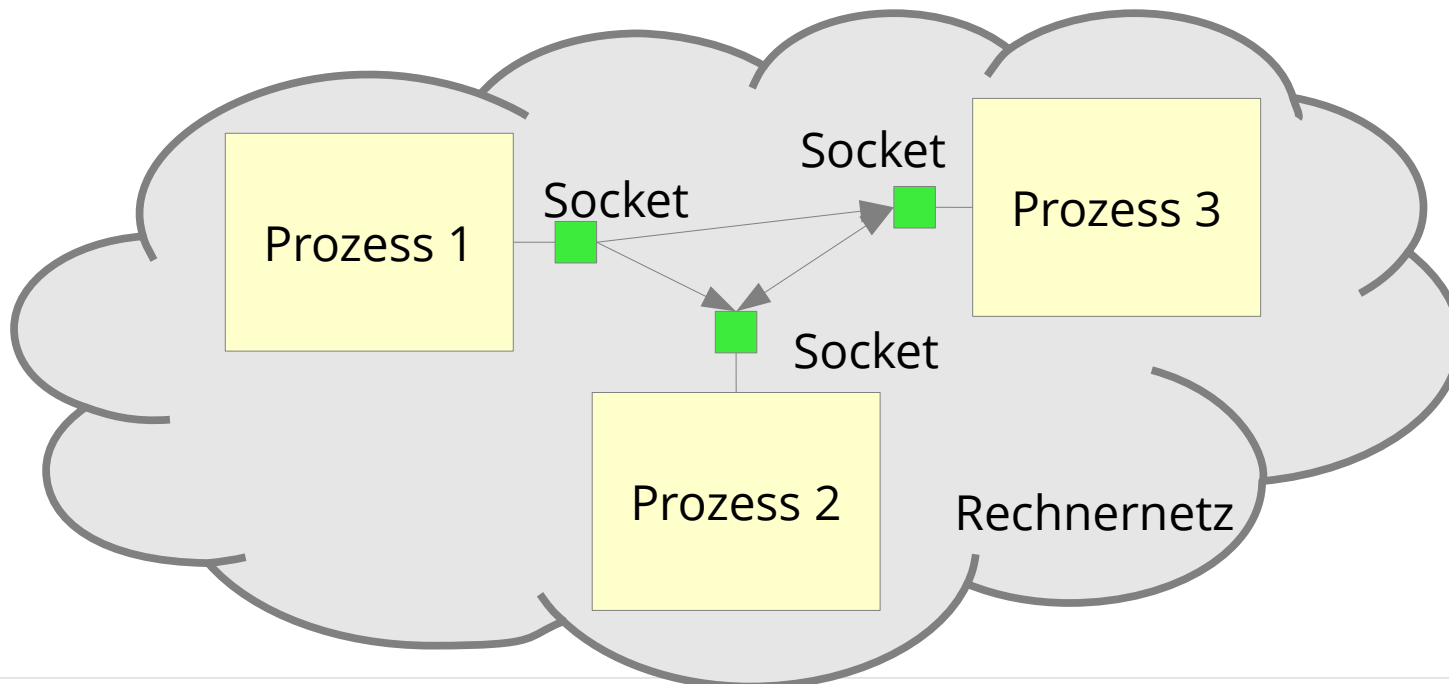
# Inhalt

- Wiederholung
- Grundlagen der Interprozesskommunikation
- Lokale Interprozesskommunikation unter UNIX
  - Signale
  - *Pipes*
  - *Message Queues*
- **Rechnerübergreifende Interprozesskommunikation**
  - *Sockets*
  - Entfernte Prozeduraufrufe (RPCs)
- Zusammenfassung



# Sockets

- Allgemeine Kommunikationsendpunkte im Rechnernetz
  - Bidirektional
  - Gepuffert
- Abstrahiert von Details des Kommunikationssystems
  - Beschrieben durch **Domäne** (Protokollfamilie), **Typ** und **Protokoll**



# Sockets: Domänen

- **UNIX Domain**
  - UNIX *Domain Sockets* verhalten sich wie bidirektionale *Pipes*.
  - Anlage als Spezialdatei im Dateisystem möglich.
- **Internet Domain**
  - Dienen der rechnerübergreifenden Kommunikation mit Internet-Protokollen
- *Appletalk Domain, DECnet Domain, ...*
- Domänen bestimmen mögliche Protokolle
  - z.B. *Internet Domain*: TCP/IP oder UDP/IP
- Domänen bestimmen die Adressfamilie
  - z.B. *Internet Domain*: IP-Adresse und *Port*-Nummer

# Sockets: Typ und Protokoll

- Die wichtigsten Sockettypen:
    - stromorientiert, verbindungsorientiert und gesichert
    - nachrichtenorientiert und ungesichert
    - nachrichtenorientiert und gesichert
  - Protokolle der Internet Domain:
    - **TCP/IP**-Protokoll
      - strom- und verbindungsorientiert, gesichert
    - **UDP/IP**-Protokoll
      - nachrichtenorientiert, verbindungslos, ungesichert
      - Nachrichten können verloren oder dupliziert werden
      - Reihenfolge kann durcheinander geraten
      - Paketgrenzen bleiben erhalten (Datagramm-Protokoll)
- Protokollangabe ist oft redundant

# Sockets: Programmierung

- Anlegen von Sockets
  - Generieren eines Sockets mit (Rückgabewert ist ein Filedeskriptor)  
`int socket(int domain, int type, int proto);`
  - Adresszuteilung
    - Sockets werden ohne Adresse generiert
    - Adressenzuteilung erfolgt durch:  
`int bind (int socket,  
          const struct sockaddr *address,  
          socklen_t address_len);`
    - `struct sockaddr_in` (für die Internet-Adressfamilie) enthält:  
`sin_family:` AF\_INET  
`sin_port:` 16-Bit-Portnummer  
`sin_addr:` Struktur mit der IP-Adresse,  
z.B. `141.76.48.99`

Hinweis: Für **IPv6** gibt es `sockaddr_in6` und `AF_INET6`

# Sockets: Programmierung

- ***Datagram Sockets***

- Kein Verbindungsaufbau notwendig

- Datagramm senden:

```
ssize_t sendto(int socket, const void *message,  
              size_t length, int flags,  
              const struct sockaddr *dest_addr,  
              socklen_t dest_len);
```

- Datagramm empfangen:

```
ssize_t recvfrom(int socket, void *buffer,  
               size_t length, int flags,  
               struct sockaddr *address,  
               socklen_t *address_len);
```

# Sockets: Programmierung

- **Stream Sockets**

- Verbindungsaufbau notwendig
- *Client* (Benutzer, Benutzerprogramm) will zu einem *Server* (Dienstanbieter) eine Kommunikationsverbindung aufbauen

- **Client:** Verbindungsaufbau bei stromorientierten Sockets

- Verbinden des Sockets mit

```
int connect (int socket,  
            const struct sockaddr *address,  
            socklen_t address_len) ;
```

- Senden und Empfangen mit **write** und **read** (oder **send** und **recv**)
- Beenden der Verbindung mit **close** (schließt den Socket)

# Sockets: Programmierung

- *Server*: akzeptiert Anfragen/Aufträge
  - bindet *Socket* an eine Adresse (sonst nicht erreichbar)
  - bereitet *Socket* auf Verbindungsanforderungen vor durch

```
int listen(int s, int queuelen);
```
  - akzeptiert einzelne Verbindungsanforderungen durch

```
int accept(int s, struct sockaddr *addr,  
          socklen_t *addrlen);
```

    - gibt einen neuen Socket zurück, der mit dem Client verbunden ist
    - blockiert, falls kein Verbindungswunsch vorhanden
  - liest Daten mit **read** und führt den angebotenen Dienst aus
  - schickt das Ergebnis mit **write** zurück zum *Client*
  - schließt den neuen Socket mit **close**

# Sockets: Programmierung

```
#define PORT 6789
#define MAXREQ (4096*1024)

char buffer[MAXREQ], body[MAXREQ+1000], msg[MAXREQ+2000];

void error(const char *msg) { perror(msg); exit(1); }

int main() {
    int sockfd, newsockfd;
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;
    int n;
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) error("ERROR opening socket");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(PORT);
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error("ERROR on binding");
    listen(sockfd,5);
    ...
}
```

Hier wird der Socket erstellt und an eine Adresse gebunden.



# Sockets: Programmierung

```
...
while (1) {
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0) error("ERROR on accept");
    bzero(buffer, sizeof(buffer));
    n = read(newsockfd, buffer, sizeof(buffer)-1);
    if (n < 0) error("ERROR reading from socket");
    snprintf (body, sizeof (body),
              "<html>\n<body>\n"
              "<h1>Hallo Webbrowser</h1>\nDeine Anfrage war ... \n"
              "<pre>%s</pre>\n"
              "</body>\n</html>\n", buffer);
    snprintf (msg, sizeof (msg),
              "HTTP/1.0 200 OK\n"
              "Content-Type: text/html\n"
              "Content-Length: %zd\n\n%s", strlen (body), body);
    n = write(newsockfd, msg, strlen(msg));
    if (n < 0) error("ERROR writing to socket");
    close(newsockfd);
}
}
```

Eine neue Verbindung akzeptieren

HTTP-Anfrage einlesen

Antwort generieren und zurückschicken

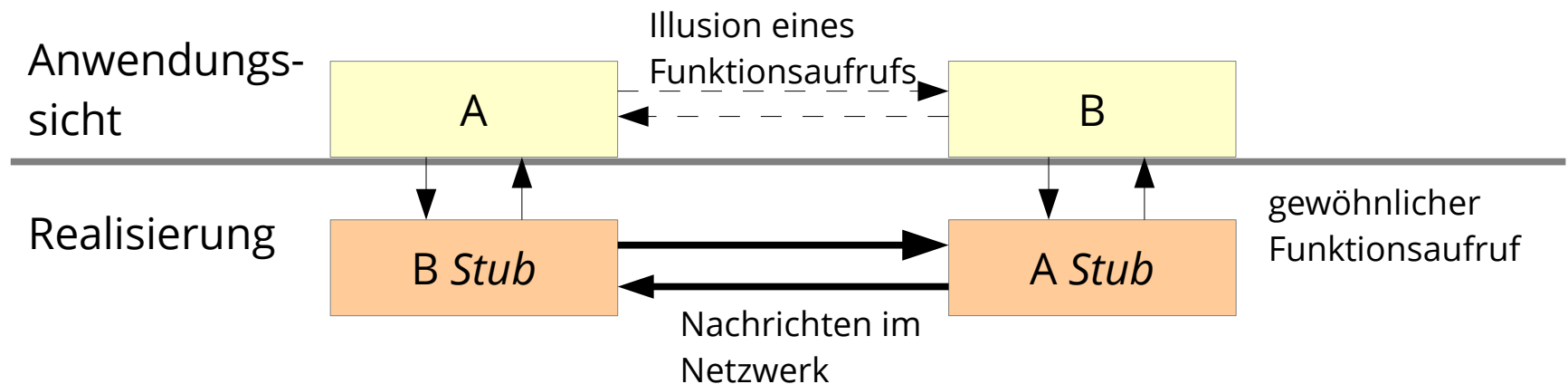
Verbindung wieder schließen.

# Sockets: Beispiel HTTP Echo



# Fernaufruf (RPC)

- Funktionsaufruf über Prozessgrenzen hinweg (*Remote Procedure Call*)
  - hoher Abstraktionsgrad
  - selten wird Fernaufruf direkt vom System angeboten; benötigt Abbildung auf andere Kommunikationsformen (z.B. auf Nachrichten)
  - Abbildung auf mehrere Nachrichten
    - Auftragsnachricht transportiert Aufrufabsicht und Parameter.
    - Ergebnismnachricht transportiert Ergebnisse des Aufrufs.



- **Beispiele:** NFS (ONC RPC), Linux D-BUS, Java RMI

# Inhalt

- Wiederholung
- Grundlagen der Interprozesskommunikation
- Lokale Interprozesskommunikation unter UNIX
  - Signale
  - *Pipes*
  - *Message Queues*
- Rechnerübergreifende Interprozesskommunikation
  - *Sockets*
  - Entfernte Prozeduraufrufe (RPCs)
- **Zusammenfassung**

# Zusammenfassung

- Es gibt zwei Arten der Interprozesskommunikation:
  - **nachrichtenbasiert**
    - die Daten werden kopiert
    - geht auch über Rechengrenzen
  - **über gemeinsamen Speicher**
    - (war heute nicht dran)
- UNIX-Systeme bieten verschiedene Abstraktionen
  - Signale, *Pipes*, *Sockets*, *Message Queues*
  - Insbesondere die *Sockets* werden häufig verwendet.
    - Ihre Schnittstelle wurde standardisiert.
    - Praktisch alle Vielzweckbetriebssysteme implementieren heute *Sockets*.