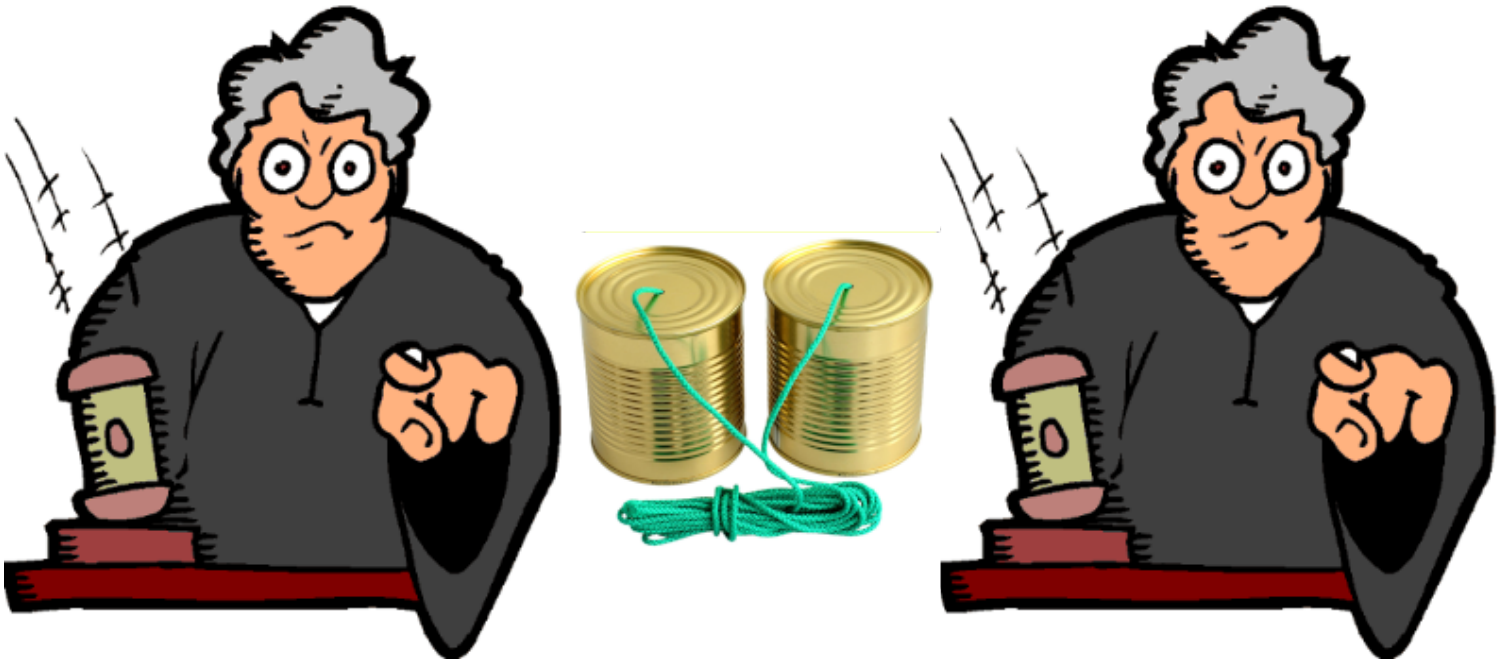
Betriebssysteme

Robert Kaiser

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022



https://www.animierte-gifs.net/data/media/1798/animiertes-richter-bild-0001.gif

Prozesskommunikation

Hochschule **RheinMain**

1. Ereigniskommunikation
2. Nachrichtenkommunikation
3. Zusammenfassung

Ereigniskommunikation

Bisher:

- Prozesse kooperieren durch gemeinsame Benutzung von Betriebsmitteln (z.B. Speicherbereichen) (Sharing-Paradigma)
- Synchronisationsmechanismen zur Vermeidung von zeitkritischen Abläufen
- Synchronisationsmechanismen auch zur Signalisierung von Ereignissen zwischen Prozessen geeignet (z.B. Fertig-Signale zur Durchsetzung einer Vorrangrelation zwischen Prozessen eines Prozesssystems)

Hier:

- Betriebssysteme können darüberhinaus spezielle Mechanismen für die Kommunikation (Signalisierung) von Ereignissen anbieten. (Diese Mechanismen können wiederum auch zur Synchronisation eingesetzt werden).
- Kommunikation von Ereignissen ist ein Spezialfall der Nachrichten-orienterten Kommunikation (vgl. 6.2).

Ereignisse

- Ereignisse zeigen das Eintreten einer „wichtigen Begebenheit“ an, z.B.:
  - das Erreichen eines bestimmten Zustands
  - das Eintreten eines Fehlers
  - den Ablauf einer vorgegebenen Zeitspanne.
- Ereigniskonzept als Übertragung des Hardware-Interrupt-Konzepts auf die Software-Ebene
- Ereignisse werden lediglich nummeriert, Zuordnung einer tatsächlichen Bedeutung kann ganz oder teilweise vorgegeben oder den Anwendungen überlassen sein.
- Bereitstellung eines effizienten Ereigniskonzepts ist insbesondere für Echtzeitbetriebssysteme von Bedeutung:
  In Echtzeitanwendungen modellieren Ereignisse häufig reale Ereignisse in der Außenwelt.

Signale in UNIX (1)

Hochschule RheinMain

Eigenschaften:

- UNIX unterscheidet zwei Arten von Signalen:
  - (Synchrone) Fehler-Ereignisse, die bei der Programmausführung auftreten, z.B.:
    - Adressierung eines ungültigen Speicherbereichs ($\rightarrow$ SIGSEGV)
    - Versuch der Ausführung einer ungültigen Instruktion ($\rightarrow$ SIGILL)
  - Asynchrone Ereignisse, die von außerhalb des Prozesses stammen, z.B.:
    - Beendigung eines Kind-Prozesses ($\rightarrow$ SIGCHLD)
    - Terminalleitungstreiber zeigt Ende der Übertragung an ($\rightarrow$ SIGHUP)
    - Programmierter Timer läuft ab ($\rightarrow$ SIGALRM)
- Das UNIX-Signalkonzept überträgt des Hardware-Interrupt-Konzepts auf die Software-Ebene:

Interrupt	$\widetilde{=}$	Signal
Interrupt Handler	$\widetilde{=}$	Signal Handler
Maskieren von Interrupts	$\widetilde{=}$	Maskieren von Signalen

Signale in UNIX (2)

- UNIX unterscheidet eine feste Anzahl von Signalen
  - Linux: `SIG_MAXSIG` = 64
  - SysV: `MAXSIG` = 32

  i.d.R. definiert durch Wortlänge → Signalmenge darstellbar als Bitmaske
- Die Bezeichnung der Signale und ihr zugehöriger Zahlenwert sind z.T. für verschiedene UNIX Versionen (Linux, BSD, SysV) unterschiedlich.
- Typische Signale (Linux, komplette Liste mit: `kill -l`):

Signal	Wert	Bedeutung	Default-Aktion
SIGHUP	1	Hangup, Verbindungsabbruch	exit
SIGILL	4	ungültiger Maschinenbefehl	core dump
SIGABRT	6	Abbruch des Prozesses mit Dump	core dump
SIGFPE	8	Gleitkommafehler	core dump
SIGKILL	9	Kill-Signal, **nicht abfangbar**	exit
SIGUSR1	10	Anwender-definierbar 1	exit
SIGSEGV	11	ungültige Speicheradresse	core dump
SIGUSR2	12	Anwender-definierbar 2	exit
SIGPIPE	13	Schreiben in Pipe ohne Empfänger	exit (vgl. 6.2)
SIGALRM	14	Ablauf des Weckers	exit
SIGTERM	15	Aufforderung zur Terminierung	exit
SIGCHLD	17	Statusänderung Kindprozess	ignore
SIGCONT	18	Fortsetzung	ignore
SIGTSTP	20	Anwender-Stop von tty	stop

Signale in UNIX (3)

- Ein Prozess im Zustand **rechnend** kann durch den BS-Kern Signale **senden und empfangen**.
- **Nicht-rechnende** Prozesse können nur Signale von rechnenden Prozessen **empfangen**.
- Ein Prozess kann festlegen, dass er bestimmte Signale selbst durch einen **Signal-Handler** behandeln oder **ignorieren** will (s.u.). Einige Signale (wie z.B. `SIGKILL`) können jedoch nicht abgefangen werden.
- Ein Prozess kann nur Signale an Prozesse mit **derselben** realen oder effektiven **User-Id** senden (Prozessgruppe), insbesondere an sich selbst. Darüber hinaus können Prozesse mit der realen oder effektiven **User-Id 0** (d.h. mit Superuser-Berechtigung) Signale an **beliebige Prozesse** senden.
- Der BS-Kern kann jedes Signal an jeden Prozess senden.
- Die Verarbeitung von anstehenden Signalen geschieht, wenn der Prozess aktiv ist und im Kernmodus ausgeführt wird.
- Prozesse können Signale **maskieren**. Die Signalzustellung wird dann vorübergehend ausgesetzt.
- Das Betriebssystem hält **Signalmasken** im Prozesskontrollblock des Prozesses (vgl. 2.1).

Systemdienste zur Signalisierung (1)

- `#include <signal.h>`
- `int kill(pid_t pid, int sig);`

  Sende das Signal `sig` an den Prozess `pid` (Mitglied derselben Prozessgruppe) oder an alle Prozesse dieser Gruppe, falls `pid` Null ist. Weitere Möglichkeiten für Prozesse mit User-Id 0 (Superuser).

- `int sigpause(int sig);`

  Das Signal `sig` wird aus der Signalmaske entfernt (d.h. zugelassen), und der Prozess blockiert, bis ein Signal empfangen oder der Prozess beendet wird.

- `int pause(void);`

  Der Prozess blockiert, bis ein Signal empfangen oder der Prozess beendet wird.

- `int sighold(int sig);`

  Das Signal `sig` wird der Signalmaske hinzugefügt und dadurch bis auf weiteres zurückgehalten (maskiert).

Beispiel: Programmstart mit Timeout-Abbruch


```
$ ./timeoutdemo /bin/sleep 4
Kind pid=28503 gestartet
Kind endet, Status=0
```

```
#include <...>
#define TIMEOUT 5
int main(int argc, char *argv[]) {
^^Iint pid, status;
^^Iif(pid=fork()) {
^^I^^Ifprintf(stderr,"Kind pid=%d gestartet\n",pid);
^^I^^Isleep(TIMEOUT);
^^I^^Iif(waitpid(pid,&status,WNOHANG)==0) {
^^I^^I^^Ifprintf(stderr,"Timeout, Abbruch!\n");
^^I^^I^^Ikill(pid, SIGKILL);
^^I^^I^^Iwait(&status);
^^I^^I}
^^I^^Ifprintf(stderr,"Kind endet, Status=%d\n",status);
^^I} else {
^^I^^Iexecve(argv[1], argv+1, NULL);
^^I^^Ifprintf(stderr,"Fehler beim Starten von %s",argv[1]);
^^I^^Iexit(EXIT_FAILURE);
^^I}
^^Ireturn EXIT_SUCCESS;
}
^^I^^I
```

- Kind beendet sich nach weniger als TIMEOUT Sekunden: OK
- Kind läuft länger als TIMEOUT Sekunden ⇒ Abbruch mit `kill()`

Beispiel: Programmstart mit Timeout-Abbruch


```
$ ./timeoutdemo /bin/sleep 6
Kind pid=28407 gestartet
Timeout, Abbruch!
Kind endet, Status=9
```

```
#include <...>
#define TIMEOUT 5
int main(int argc, char *argv[]) {
^^Iint pid, status;
^^Iif(pid=fork()) {
^^I^^Ifprintf(stderr,"Kind pid=%d gestartet\n",pid);
^^I^^Isleep(TIMEOUT);
^^I^^Iif(waitpid(pid,&status,WNOHANG)==0) {
^^I^^I^^Ifprintf(stderr,"Timeout, Abbruch!\n");
^^I^^I^^Ikill(pid, SIGKILL);
^^I^^I^^Iwait(&status);
^^I^^I}
^^I^^Ifprintf(stderr,"Kind endet, Status=%d\n",status);
^^I} else {
^^I^^Iexecve(argv[1], argv+1, NULL);
^^I^^Ifprintf(stderr,"Fehler beim Starten von %s",argv[1]);
^^I^^Iexit(EXIT_FAILURE);
^^I}
^^Ireturn EXIT_SUCCESS;
}
^^I^^I
```

- Kind beendet sich nach weniger als TIMEOUT Sekunden: OK
- Kind läuft länger als TIMEOUT Sekunden ⇒ Abbruch mit `kill()`

Systemdienste zur Signalisierung (2)

- `int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);`

  Zuordnung einer Aktion bei Auftreten des Signals `sig` und Speichern der bisher gültigen Aktion gemäß POSIX. Erfolgt i.d.R. nur einmal im Programm für jedes Signal, für das die Default-Aktion geändert werden soll.
- Dabei ist:

```
struct sigaction {
   int sa_flags;          /* Flags, POSIX: nur SA_NOCLDSTOP */
   void (*sa_handler)(); /* Adresse einer eigenen Funktion */
                          /* oder SIG_IGN oder SIG_DFL      */
   sigset_t sa_mask;      /* zu maskierende Signale, wenn   */
}                         /*    handler ausgefuehrt wird.   */
```

- Fehler `EINVAL`, falls `sig` = `SIGKILL` oder `SIGSTOP`.
- Im Falle von `sig==SIGCHLD` bewirkt `SA_NOCLDSTOP`, dass `SIGCHLD` erst gesendet wird, wenn alle Kindprozesse beendet sind (Normalfall).
- Nach Abschluss der erfolgreichen Behandlung erfolgt die Fortsetzung des Programms an der unterbrochenen Stelle.

Beispiel: Signal abfangen

Hochschule RheinMain

```
#include <...>

void myIntHandler(int sig) {
^^Ifprintf(stderr,"Autsch!\n");
}

int main(void) {
^^Iint i;
^^Istruct sigaction act, old;
^^Iact.sa_handler = myIntHandler;
^^Isigaction(SIGINT, &act, &old);
^^Ifor (i=0; i < 10; i++) {
^^I^^Iprintf("Runde %d\n",i);
^^I^^Isleep(1);
^^I}
^^Isigaction(SIGINT, &old, NULL);
^^Ireturn 0;
}
^^I^^I
```

- SIGINT (Ctrl-C) wird abgefangen!

Beispiel: Signal abfangen

Hochschule RheinMain

```
#include <...>

void myIntHandler(int sig) {
^^Ifprintf(stderr,"Autsch!\n");
}

int main(void) {
^^Iint i;
^^Istruct sigaction act, old;
^^Iact.sa_handler = myIntHandler;
^^Isigaction(SIGINT, &act, &old);
^^Ifor (i=0; i < 10; i++) {
^^I^^Iprintf("Runde %d\n",i);
^^I^^Isleep(1);
^^I}
^^Isigaction(SIGINT, &old, NULL);
^^Ireturn 0;
}
^^I^^I
```

```
$ ./sigactionhandler
Runde 0
Runde 1
Runde 2
^CAutsch!
Runde 3
Runde 4
Runde 5
^CAutsch!
Runde 6
^CAutsch!
Runde 7
Runde 8
Runde 9
```

- SIGINT (Ctrl-C) wird abgefangen!

Systemdienste zur Signalisierung (3)

- `int sigrelse(int sig);`

  Das Signal `sig` wird aus der Signalmaske entfernt und damit wieder zugelassen.

- `int sigignore(int sig);`

  Das Signal `sig` wird auf die Disposition `SIG_IGN` gesetzt und damit bis auf weiteres vom aufrufenden Prozess ignoriert.

- `int alarm(int seconds);`

  Das Signal `SIGALRM` wird nach Ablauf der übergebenen Zeitdauer zugestellt („Wecker stellen“).

wird im Praktikum vertieft.

Nachrichtenorientierte Kommunikation

Motivation:

- Signalisierung von Ereignissen ist Spezialfall des Austausches von Nachrichten beliebiger Art
  (Paradigma der Kooperation durch Nachrichtenaustausch).
- Betriebssysteme besitzen i.d.R. entsprechende Mechanismen zur nachrichtenorientierten Kommunikation.
- Nachrichtenorientierte Kommunikation ist sowohl für lokale wie auch für verteilte Rechensysteme anwendbar.

Hier:

- Besprechung der Grundlagen eines solchen allgemeinen Nachrichten(austausch)systems zur Interprozesskommunikation.

Nachrichtensystem (Message System)

Aufgabe

- Mechanismus, so dass Prozesse einander Nachrichten zukommen lassen können, ohne über gemeinsamen Speicher verfügen zu müssen.

Operationen

- `SEND ( message, ... )` zum Senden einer Nachricht durch einen **Sendeprozess** oder einfach **Sender**.
- `RECEIVE ( message, ... )` zum Empfangen einer Nachricht durch einen **Empfangsprozess** oder **Empfänger**.

Kopplung/Verbindung der Kommunikationspartner

- notwendig, um die Kommunikation zu ermöglichen.
- Verbindung heißt **Kommunikationskanal** oder **Kanal**.
- Ein Prozess kann zu einem Zeitpunkt mehrere Kanäle aufrecht erhalten (i.d.R. zu verschiedenen Prozessen).

Wichtige Entwurfsaspekte eines Nachrichtensystems

- Adressierung, Pufferung, Nachrichtenstruktur

Kommunikationskanal

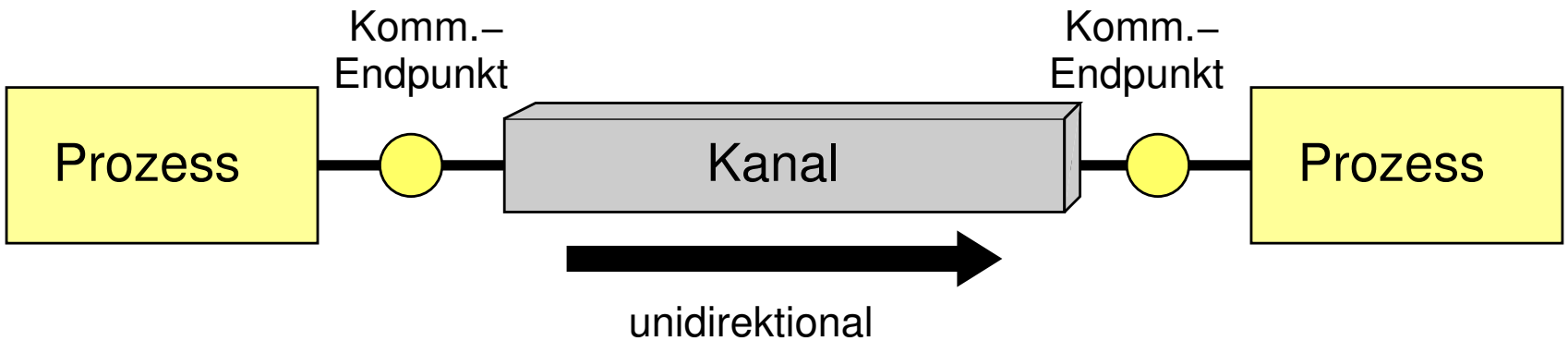
Hochschule RheinMain

Anzahl der Kommunikationsteilnehmer eines Kanals

- Genau zwei: Regelfall.
- Mehr als zwei: Gruppenkommunikation (Multicast-Dienst, Spezialfall: Broadcast, d.h. Rundsendung an alle), hier nicht weiter betrachtet.

Richtung des Nachrichtenflusses eines Kanals

- Kanal heißt **gerichtet** oder **unidirektional**, wenn ein Prozess ausschließlich die Sender-Rolle, der andere ausschließlich die Empfänger-Rolle ausübt, ansonsten **ungerichtet** oder **bidirektional**.



Adressierung (1)

Direkte Adressierung

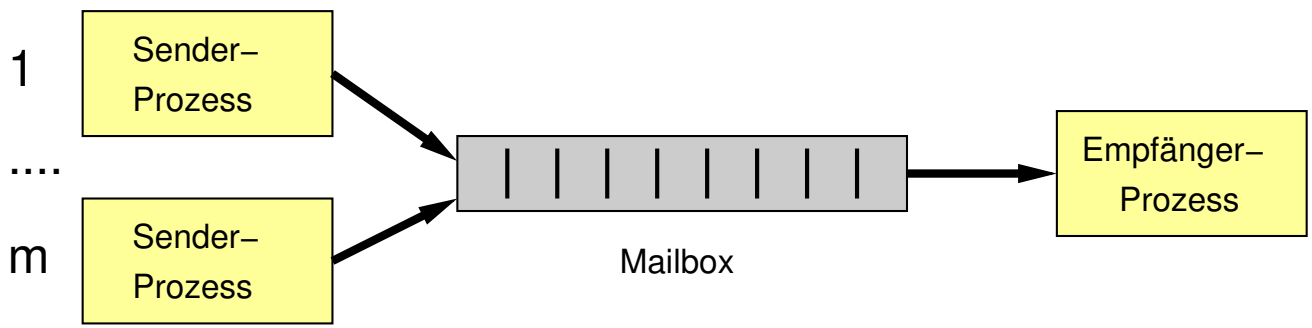
- Prozesse haben eindeutige Adressen.
- Benennung ist **explizit** und **symmetrisch**:
  sendender Prozess muss Empfänger benennen und umgekehrt.
  - `SEND ( P, message )`
    Sende eine Nachricht an Prozess `P`.
  - `RECEIVE ( Q, message )`
    Empfange eine Nachricht von Prozess `Q`.
- **Asymmetrische** Variante (z.B. für Server-Prozesse):
  - Sender benennt Empfänger, Empfänger (Server-Prozess) wird mit dem Empfang die Identität des Senders bekannt:
  - `SEND ( P, message )`
  - `RECEIVE ( sender_id , message )`

Adressierung (2)

Hochschule RheinMain

Indirekte Adressierung

- Kommunikation erfolgt indirekt über zwischengeschaltete **Mailboxes** (Puffer für eine Anzahl von Nachrichten):
  - `SEND ( mbox, message )`
    Sende eine Nachricht an Mailbox `mbox`.
  - `RECEIVE ( mbox, message )`
    Empfange eine Nachricht von Mailbox `mbox`.
- Vorteile:
  - Verbesserte Modularität.
  - Prozessmenge kann transparent restrukturiert werden, z.B. nach Ausfall eines Empfangsprozesses.
  - Erweiterte Zuordnungsmöglichkeiten von Sendern und Empfängern, wie z.B. m:1, 1:n, m:n.
- Beispiel: m:1



Pufferung

Hochschule RheinMain

Definition: Kapazität

Die **Kapazität eines Kanals** bezeichnet die Anzahl der Nachrichten, die vorübergehend in einem Kanal gespeichert werden können, um Sender und Empfänger zeitlich zu entkoppeln.

- Die Pufferungsfähigkeit eines Kanals wird i.d.R. durch einen Warteraum / Warteschlange im Betriebssystemkern erreicht.
- Möglichkeiten:
  1. Keine Pufferung (Kapazität Null)
  2. Beschänkte Kapazität
  3. Unbeschänkte Kapazität

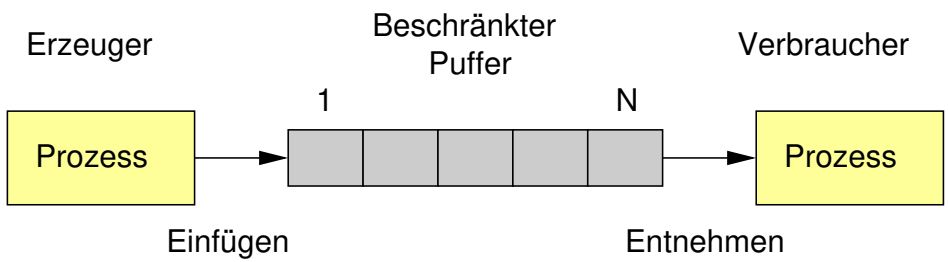
Keine Pufferung (Kapazität Null)

- Der Sender wird blockiert, wenn die `SEND`-Operation vor der entsprechenden `RECEIVE`-Operation stattfindet. Wird dann die entsprechende `RECEIVE`-Operation ausgeführt, wird die Nachricht ohne Zwischenspeicherung **unmittelbar** vom Sender- zum Empfänger-Prozess kopiert.
- Findet umgekehrt `RECEIVE` zuerst statt, so wird der Empfänger bis zum Aufruf der `SEND`-Operation blockiert.
- Diese ungepufferte Kommunikationsform, die Sender und Empfänger zeitlich sehr eng koppelt, heisst **Rendezvous** oder **synchroner Nachrichtenaustausch**.
- Beispiele:
  - Hoare: Communicating Sequential Processes (CSP).
  - Kommunikation zwischen Ada-Tasks.
  - Mikrokern „L4“ [J. Liedtke 1995. *On μ-Kernel construction*].
- Synchroner Nachrichtenaustausch wird häufig als zu inflexibel angesehen.

Beschränkte Kapazität

Hochschule RheinMain

- Der Kanal kann zu einem Zeitpunkt maximal `N` Nachrichten enthalten (Warteraum der Kapazität `N`).
- Im Falle einer `SEND`-Operation bei nicht-vollem Warteraum wird die Nachricht im Warteraum abgelegt, und der Sendeprozess fährt fort.
- Ist der Warteraum voll (enthält `N` gesendete aber noch nicht empfangene Nachrichten), so wird der Sender blockiert, bis ein freier Warteplatz vorhanden ist.
- Das Ablegen der Nachricht kann durch Kopieren der Nachricht oder Speichern eines Zeigers auf die Nachricht realisiert sein.
- Analog wird der Empfänger bei Ausführung einer `RECEIVE`-Operation blockiert, wenn der Warteraum leer ist.
- Vgl. 5.3.1 (Erzeuger-Verbraucher-Problem):



Unbeschränkte Kapazität

- Der Kanal kann potenziell eine unbeschränkte Anzahl von Nachrichten enthalten.

→ Die `SEND`-Operation **kann nicht blockieren**.

- Lediglich der Empfänger kann bei Ausführung einer `RECEIVE`-Operation blockieren, wenn der Warteraum leer ist.
- Implementierung ist möglich, indem Sender und Empfänger die Warteplätze beim Aufruf „mitbringen“.

Pufferung: Konsequenzen

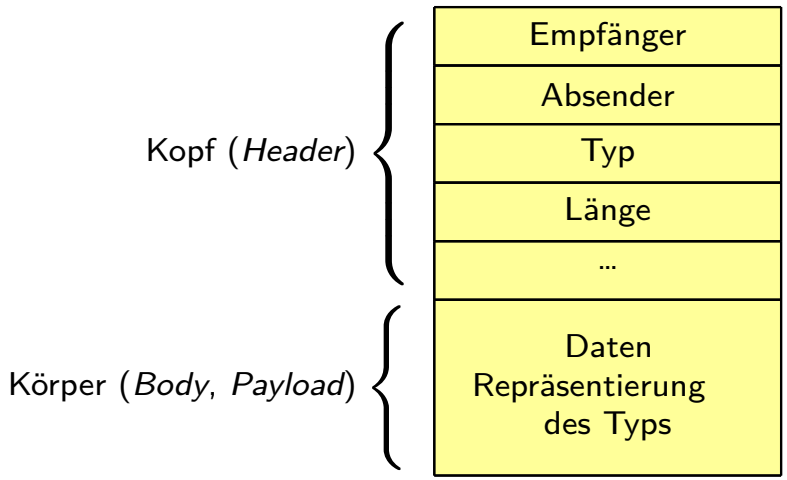
- Gepufferte Kommunikation (beschränkt oder unbeschränkt) bewirkt zeitlich lose Kopplung der Kommunikationspartner.
- Nach Abschluss der SEND-Operation weiß der Sender nicht, ob der Empfänger die Nachricht erhalten hat. Er kennt i.d.R. auch keine maximale Zeitdauer dafür.
- Wenn dieses Wissen wesentlich für den Sender ist, muss dazu eine explizite Kommunikation zwischen Sender und Empfänger durchgeführt werden:

Prozess P (Sender):		Prozess Q (Empfänger):
…		…
`SEND( Q, message);`	$\longrightarrow$	`RECEIVE( P, message);`
`RECEIVE( Q, reply);`	$\longleftarrow$	`SEND( P, "acknowledge");`
…		…

Semantik von Nachrichten (1)

Typisierte Nachrichten

- Nachrichten haben eine typisierte Struktur.
- Typ ist Sender und Empfänger und z.T. dem Nachrichtensystem bekannt und wird in Operationen verwendet.
- Beispielhafter Aufbau einer Nachricht:



Semantik von Nachrichten (2)

Nachrichtencontainer

- Nachrichten sind für Sender und Empfänger identifizierbare Einheiten fester oder variabler Länge. Nachrichtengrenzen bleiben erhalten.
- Korrekte Interpretation der internen Struktur einer Nachricht obliegt den Kommunikationspartnern.
- Beispiel: UNIX message queues.

Bytestrom

- Empfänger (und das Nachrichtensystem) sehen ausschließlich eine Folge von Zeichen (Bytestrom).
- Übergebene Nachrichten verschiedener SEND-Operationen sind als Einheiten nicht mehr identifizierbar. Nachrichtengrenzen gehen verloren.
- Beispiel: UNIX pipes.

Implementierung

Klassische Implementierung

- Speicher für zu puffernde Nachrichten liegt im BS-Kern.
- Bei Ausführung von `SEND` wird die Nachricht aus dem sendenden Prozess(-adressraum) in den BS-Kern kopiert.
- Bei Ausführung von `RECEIVE` wird die Nachricht aus dem BS-Kern in den empfangenden Prozess(-adressraum) kopiert.

→ 2-maliges Kopieren. Aufwändig!

Integration mit Speicherverwaltung

- In modernen Betriebssystemen wird die Implementierung von Nachrichtenkommunikation mit dem Virtual Memory Management integriert (siehe Kap. 8, z.B. Mach).
- Versenden von Nachrichten wird auf das Kopieren von Deskriptoren von Speicherbereichen beschränkt.
- Die Speicherbereiche werden durch Markierung als *copy-on-write* nur im Notfall wirklich kopiert (lazy copying, vgl. Kap. 8).

Beispiel: UNIX (1)

Hochschule RheinMain

UNIX stellt mehrere Dienste zum Nachrichtenaustausch zwischen Prozessen zur Verfügung:

- (Anonyme) Pipes
  - Ursprünglicher Mechanismus zum **uni**direktionalen Nachrichtentransport (Bytestrom) zwischen verwandten Prozessen
  - Vererbung der Kommunikationsendpunkte durch fork().
- Named Pipes oder FIFOs
  - Erweiterung auf nicht-verwandte Prozesse.
  - Dateisystem-Namensraum als gemeinsamer Namensraum zur Benennung von Named Pipes.
  - Repräsentierung von Named Pipes im Dateisystem.
- Message Queues
  - Bestandteil der System V IPC-Mechanismen.
  - Message Queue ist komplexes, gemeinsam benutztes Objekt zum Austausch typisierter Nachrichten zwischen potentiell beliebig vielen Sende- und Empfangs-Prozessen.

Beispiel: UNIX (2)

- Sockets
  - In 4.2BSD UNIX zusammen mit TCP/IP eingeführtes Konzept zur allgemeinen, rechnerübergreifenden, bidirektionalen, nachrichtenorientierten Interprozesskommunikation.
  - zur Programmierung von Client/Server-Kommunikation geeignet.
  - weit verbreitet (wird in LV „Verteilte Systeme“ behandelt).
- STREAMS
  - Mit UNIX System V Rel. 3 erstmals eingeführte Gesamtumgebung zur Entwicklung von geschichteten IPC-Protokollen.
  - Modularisierung durch verkettbare STREAMS-Module.
  - In UNIX System V Rel. 4 werden Pipes, Named Pipes, Terminal-Protokolle und Netzwerkkommunikation in der STREAMS-Umgebung bereitgestellt.

Beispiel: Anonyme Pipes

Eigenschaften:

- Anonyme Pipe definiert namenlosen unidirektionalen Kommunikationskanal, über den zwei Prozesse mit einem gemeinsamen Vorfahren einen Bytestrom kommunizieren können.
- Pipe besitzt eine Pufferkapazität von einer Seite (z.B. 4 Kb)
- wird von den Prozessen wie eine Datei behandelt.

Typischer Umgang:

- Elternprozess legt Pipe an, erzeugt zwei Kindprozesse durch `fork()`, die die geöffneten Enden der Pipe als File-Deskriptoren erben.
- Der Schreiber schließt (mittels `close()`) die Lese-Seite, der Leser die Schreib-Seite, der Elternprozess beide Seiten.
- Analog kann der Elternprozess mit einem Kindprozess über eine Pipe kommunizieren.

Anonyme Pipes: Systemdienste

- `int pipe(int fds[2])`

  Erzeugen einer anonymen geöffneten Pipe. Der File-Deskriptor `fds[0]` kann zum Lesen, `fds[1]` zum Schreiben benutzt werden.

- `int write(int fd, char* buf, unsigned length)`

  Senden von `length` bytes in die durch `fd` bezeichnete Pipe. Aufrufer blockiert, falls Pipe voll ist. Nicht-blockierendes Schreiben ist nach `fcntl()` mit `O_NDELAY` möglich. Write erzeugt `SIGPIPE`-Signal, falls Leser-Seite geschlossen wurde.

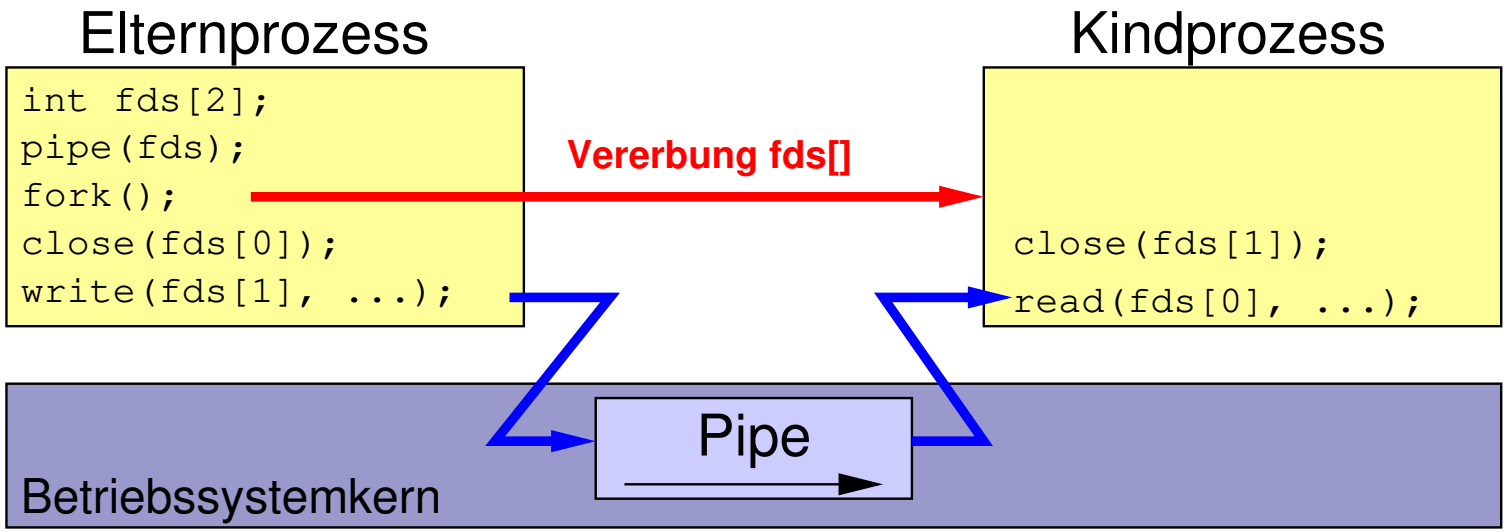
- `int read(int fd, char* buf, unsigned length)`

  Empfangen von maximal `length` Bytes aus der durch `fd` bezeichneten Pipe. Aufrufer blockiert, falls Pipe leer ist. (Blockierung kann analog zu `write()` vermieden werden). Rückgabewert ist die tatsächlich gelesene Anzahl Bytes, 0 bei Schließen der Pipe durch die Senderseite (Dateiende), -1 bei Fehler.

- `int close(int fd)`

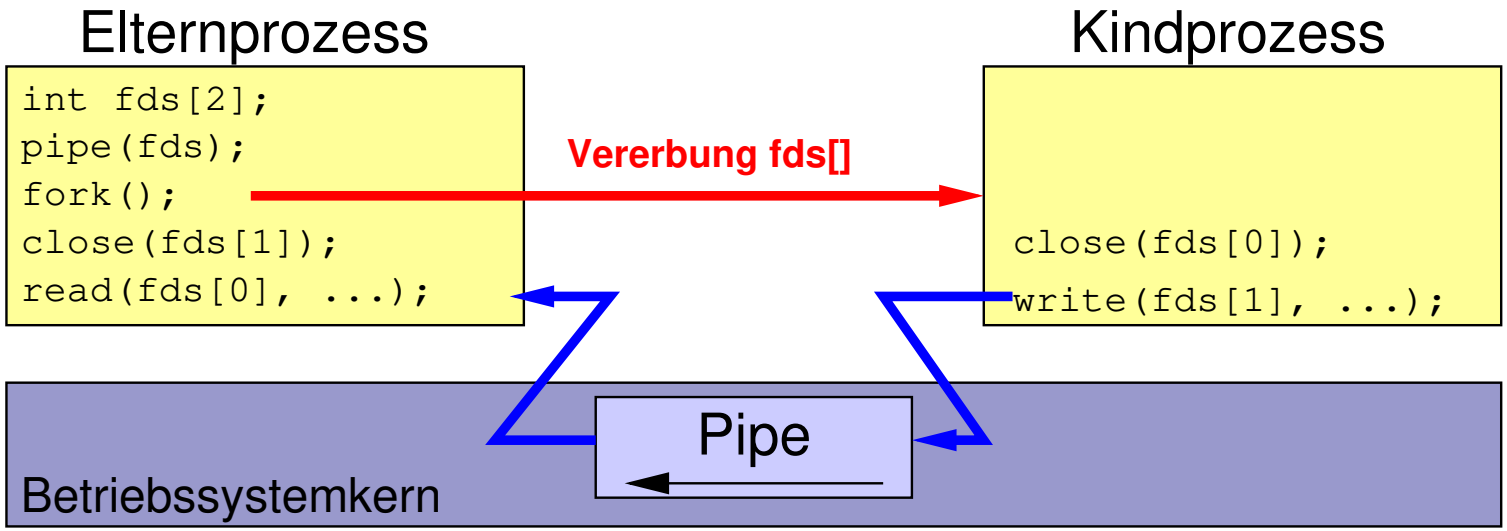
  Schließen eines Endes der Pipe zum Beenden der Kommunikation.

Beispiel: Eltern↔Kind Kommunikation



- `pipe(int fds[2])` erzeugt **zwei** File-Deskriptoren:
  - `fds[0]` ist zum Lesen geöffnet
  - `fds[1]` zum Schreiben
- Umkehrung: Kind→Eltern Kommunikation
- Fehlerbehandlung (hier weggelassen):
  Rückgabewert: 0 für OK, -1 für Fehler

Beispiel: Eltern↔Kind Kommunikation



- `pipe(int fds[2])` erzeugt **zwei** File-Deskriptoren:
  - `fds[0]` ist zum Lesen geöffnet
  - `fds[1]` zum Schreiben
- Umkehrung: Kind→Eltern Kommunikation
- Fehlerbehandlung (hier weggelassen):
  Rückgabewert: 0 für OK, -1 für Fehler

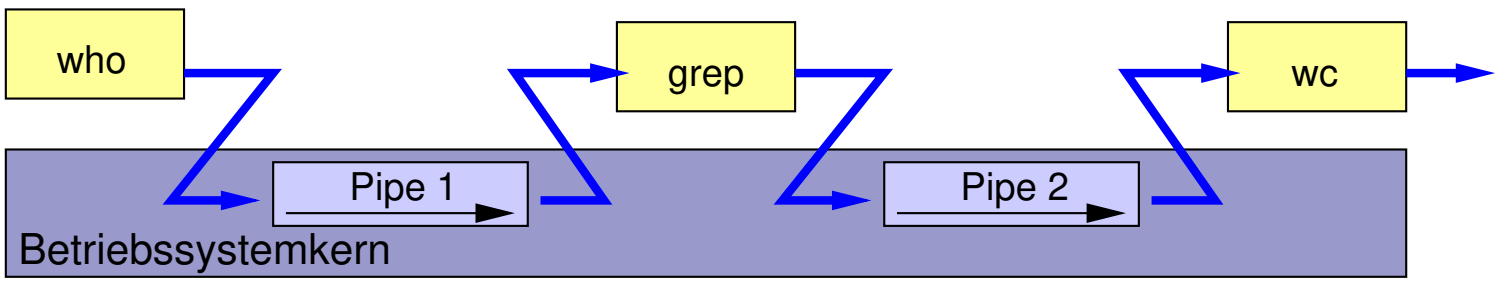
Beispiel: Pipes in der Shell

Hochschule RheinMain

- Wie oft ist „hansl“ auf dem Rechner angemeldet?
  → Shell-Kommandozeile:

```
who | grep "hansl" | wc -l
```

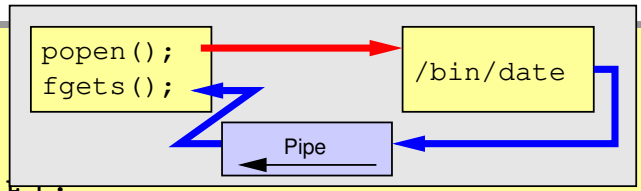
- Dazu erzeugt die Shell 2 Pipes und 3 Kindprozesse, deren Standardein-/-ausgabe-File-Deskriptoren sie wie folgt setzt:



Beispiel: popen()

Hochschule RheinMain

```
#include <stdio.h>
#include <stdlib.h>
#define MAXZEILE 80
int main(void) {
  char zeile[MAXZEILE];
  FILE *fp;
  if ( (fp = popen("/bin/date", "r")) == NULL) {
    perror("Fehler bei popen"); exit(1);
  }
  if (fgets(zeile, MAXZEILE, fp)== NULL) {
    perror("Fehler bei fgets"); exit(2);
  }
  pclose(fp);
  printf("Ausgabe von 'date' ist: %s\n", zeile);
  return 0;
}
```



⇒ Ausgabe: `Ausgabe von 'date' ist: Di 5. Jan 14:10:47 CET 2021`

- `popen()` startet ein Kommando als Subprozess, in dessen Standardeingabe geschrieben ("w") **oder** dessen Standardausgabe gelesen ("r") werden kann (entweder/oder)
- Pipe ist dabei als „Stream" (Typ `FILE*`) repräsentiert
  → Verwendung mit Stream-Funktionen (`fprintf()`, `fgets()`, ...)
- Schließen mit `pclose()`

Named Pipes oder FIFOs

Eigenschaften

- Eine benannte Pipe, auch FIFO genannt, definiert einen Kommunikationskanal, über den mehrere Sender- und mehrere Empfänger-Prozesse, die nicht miteinander verwandt sein müssen, einen gemeinsamen Bytestrom kommunizieren können.
- Eine benannte Pipe besitzt einen Namen aus dem Dateinamensraum und eine Repräsentierung im Dateisystem (*inode*, Dateikontrollblock, vgl. Kap. 10) aber keine Datenblöcke, sondern lediglich einen Seiten-Puffer im Arbeitsspeicher.
- Für benannte Pipes existieren Zugriffsrechte wie bei Dateien, die beim Öffnen überprüft werden.
- In System V sind FIFOs durch ein spezielles *vnode*-Dateisystem `fifofs` implementiert (vgl. Kap. 10).

Named Pipes: Systemdienste

- `int mknod(char* pfad, int modus, int dev)`

  Mit `mknod()` können allgemein beliebige Knoten im Dateisystem angelegt werden. Zum Erzeugen einer Named Pipe gibt `pfad` den (Datei-)Namen für das FIFO an, `modus` wird gebildet durch `S_IFIFO|<rechte>`, wobei die Bitmaske `<rechte>` die Lese-/Schreibrechte für user/group/others wie üblich kodiert (z.B. dürfen mit `0666` beliebige Prozesse lesen und schreiben), `dev` hat für FIFOs den Wert 0.
  (In SysV.4 existiert zusätzlich `mkfifo()`).

- `int open(char* pfad, int modus)`

  Öffnen der Pipe mit Namen `pfad` zum Lesen (modus `O_RDONLY`) oder Schreiben (modus `O_WRONLY`) wie bei üblicher Datei. Öffnen blockiert, bis sowohl ein Leser als auch ein Schreiber vorhanden sind. Durch Setzen des Flags `O_NDELAY` wird nicht-blockierendes Lesen oder Schreiben ermöglicht.

- `read()`, `write()` und `close()` wie bei anonymen Pipes.

Wird im Praktikum vertieft.

Beispiel: Named Pipes oder FIFOs

Hochschule RheinMain

```
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void) {
^^Iint fd;
^^Imknod("myfifo",S_IFIFO|0666,0);
^^Ifd = open("myfifo", O_WRONLY);
^^Iwrite(fd, ...);
^^I...
^^I/* Schliessen des FIFO */
^^Iunlink("myfifo");
}^^I^^I^^I^^I
```

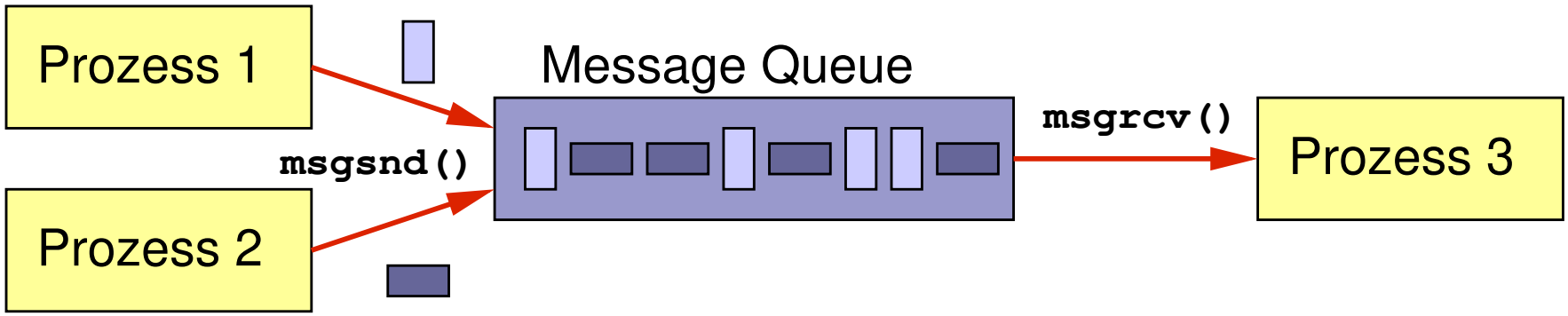
```
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void) {
^^Iint fd;
^^I...
^^Ifd = open("myfifo", O_RDONLY);
 read(fd, ...);
^^I...
}
^^I^^I^^I^^I
```

(Fehlerbehandlung weggelassen)

- `mknod()` erzeugt FIFO mit angegebenem Pfad / Zugriffsbits
- Named Pipe (FIFO) erscheint wie eine Datei im Dateibaum
- Kann daher von beliebigen Prozessen (nicht nur Parent/Child) auf dem Rechner „gesehen“ und mit den bekannten Dateioperationen genutzt werden (Zugriffsrechte vorausgesetzt)
- Schließen über Datei-Löschoperation `unlink()` (!)

Message Queues

Hochschule RheinMain



- Eine Message Queue ist eine **verkettete** (Nachrichten-) **Liste** (Nachrichtenwarteschlange), die vom Betriebssystemkern verwaltet wird.
- Empfangsreihenfolge normalerweise „first-in-first-out“, eine Priorisierung der Nachrichten ist aber auch möglich
- Im Rahmen der UNIX System V IPC-Mechanismen eingeführt (Bedeutung gesunken).
- Wie alle System V IPC-Mechanismen sind Message Queues sind persistente, systemweite Objekte.
- Shell Kommandos:
  - `ipcmk -Q` → Message Queue anlegen
  - `ipcrm -q <id>` → Message Queue löschen
  - `ipcs -q` → Message Queue Objekte anzeigen

Message Queues: Eigenschaften

- Eine Message Queue definiert einen **Kommunikationskanal**, über den **mehrere** Sender- und mehrere Empfänger-Prozesse, die **nicht miteinander verwandt** sein müssen, **typisierte** Nachrichten **variabler Länge** austauschen können.
- Message Queues besitzen einen eindeutigen Bezeichner aus dem „Key-Namensraum“.
- Eine geöffnete Message Queue wird intern über einen Message Queue Identifier (Integer) bezeichnet.
- Eine Nachricht besteht aus einem **Integer-Nachrichtentyp** und **variabel langem Nachrichteninhalt**. Festlegung der Bedeutung von Typ und Inhalt ist den Prozessen vorbehalten.
- Eine Message Queue hält Berechtigungen zur Zugriffskontrolle.
- Anzahl und Puffergröße der Message Queues werden bei der Systemgenerierung festgelegt.

Message Queues: Systemdienste

- `int msgget(key_t key, int flag)`

  Erzeugen einer neuen oder Öffnen einer existierenden Message Queue, liefert den internen Identifier der Message Queue.

- `int msgsnd(int id, struct msgbuf * buf, int msgsz, int msgflag)`

  Senden einer Nachricht an die angegebene Message Queue.

- `int msgrcv(int id, struct msgbuf * buf, int msgsz, long msgtype, int msgflag)`

  Empfangen einer Nachricht eines wählbaren oder beliebigen Typs von der angegebenen Message Queue. Der Aufrufer blockiert, falls keine Nachricht des gewünschten Typs vorhanden ist. Die Blockierung kann über ein Flag vermieden werden.

- `int msgctl(int id, int cmd, struct msqid_ds * buf)`

  Ausführung einer Kontroll-Operation auf der Message Queue zum Auslesen von Verwaltungs-Informationen, Verändern der Berechtigungen sowie zum Zerstören der Message Queue.

Zusammenfassung

Hochschule RheinMain

Was haben wir in Kap. 6 gemacht?

- Prozesse müssen kommunizieren können. Die grundlegenden Kommunikationsmöglichkeiten basieren auf der gemeinsamen Benutzung von Speicher (Sharing) sowie dem **Nachrichtenaustausch**.
- Kommunikation von Ereignissen (Signalisierung) wurde am Beispiel UNIX erläutert.
- Grundlagen der nachrichtenrichtenorientierten Kommunikation wurden vorgestellt und am Beispiel der UNIX Pipes, Named Pipes und Message Queues verdeutlicht.