Kap. 6: Prozesskommunikation

6.1 Ereigniskommunikation

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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 offerieren. (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.

Beispiel 1: Signalisierung in UNIX

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Eigenschaften:

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

Interrupt	≅ Signal
Interrupt Handler	≅ Signal Handler
Maskieren von Interrupts	≅ Maskieren von Signalen

Beispiel 1: Signalisierung in UNIX (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- UNIX unterscheidet insgesamt MAXSIG=32 Signale (Wortlänge). Die Bezeichnung der Signale und ihr zugehöriger Index sind z.T. für BSD-UNIX und UNIX System V verschieden.
- Typische Signale (UNIX System V):

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
SIGSEGV	11	ungültige Speicheradresse	core dump
SIGPIPE	13	Schreiben in Pipe ohne Empfänger	exit (vgl. 6.2)
SIGALRM	14	Ablauf des Weckers	exit
SIGTERM	15	Aufforderung zur Terminierung	exit
SIGUSR1	16	Anwender-definierbar 1	exit
SIGUSR2	17	Anwender-definierbar 2	exit
SIGCHLD	18	Statusänderung Sohnprozess	ignore
SIGTSTP	24	Anwender-Stop von tty	stop
SIGCONT	25	Fortsetzung	ignore

Beispiel 1: Signalisierung in UNIX (3)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- 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 nicht abgefangen werden.
- Ein Prozess kann nur Signale an Prozesse mit derselben realen oder effektiven User-Id senden (Prozessgruppe), insbesondere an sich selbst. Darüberhinaus 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.
- Das Betriebssystem hält die Signalmasken im Prozesskontrollblock des Prozesses (proc structure und user structure, vgl. 2.1).

Beispiel 1: Signalisierung in UNIX (4)

Systemdienste zur Signalisierung:

- `int sigaction(int sig, struct sigaction * action, struct sigaction * oldaction);`

```
#include <signal.h>
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.    */
```

- - Zuordnung einer Aktion bei Auftreten des Signals `sig` und Speichern der bisherigen Arbeitsweise gemäß POSIX (auch BSD sigvec() oder System V signal()). Erfolgt i.d.R. nur einmal im Programm für jedes Signal, für das die Default-Aktion geändert werden soll.
- - Fehler `EINVAL`, falls `sig` `SIGKILL` oder `SIGSTOP` ist.
- - Im Falle von `sig==SIGCHLD` bewirkt `SA_NOCLDSTOP`, dass `SIGCHLD` erst gesendet wird, wenn alle Sohnprozesse beendet sind (Normalfall).
- - Nach Abschluss der erfolgreichen Behandlung erfolgt die Fortsetzung des Programms an der unterbrochenen Stelle.

Beispiel 1: Signalisierung in UNIX (5)

- `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 (auch System V sigsendset()). Weitere Möglichkeiten für Prozesse mit User-Id 0 (Superuser).

- `int sigpause(int sig);`

  Das Signal `sig` wird aus der Signalmaske entfernt (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 1: Signalisierung in UNIX (6)

- `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.

- `void alarm(int seconds);`

  Das Signal `SIG_ALRM` wird nach Ablauf der übergebenen Zeitdauer zugestellt (Wecker stellen).

wird im Praktikum vertieft.

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 Interprozess-kommunikation.

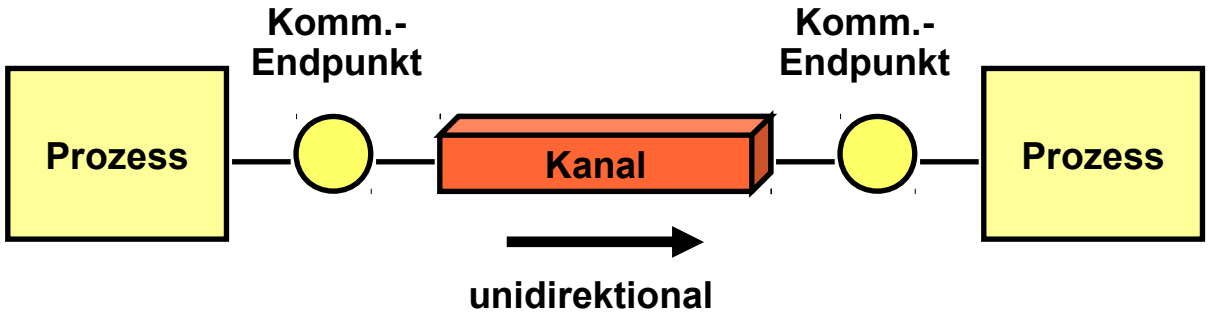
Nachrichtensystem (Message System)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Aufgabe
  - Mechanismus, so dass Prozesse einander Nachrichten zukommen lassen, 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

6.2.1 Kommunikationskanal

- 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.



6.2.2 Adressierung

- 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)

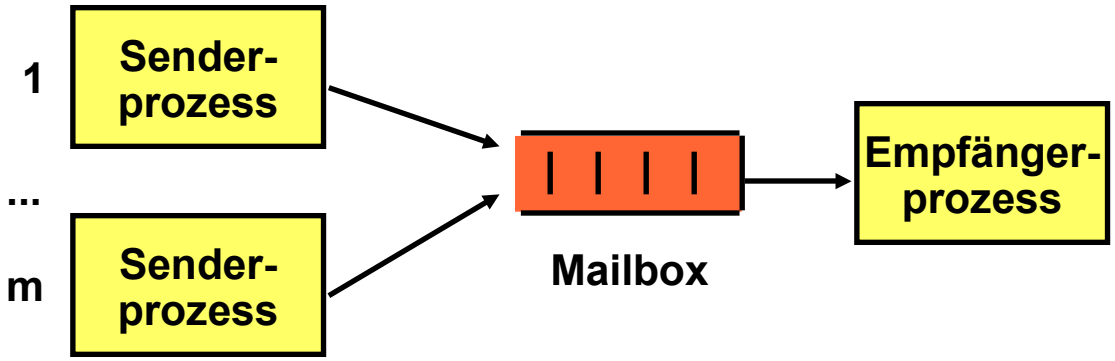
- Indirekte Adressierung
  - Kommunikation erfolgt indirekt über zwischengeschaltete Mailboxes (Puffer für eine Anzahl von Nachrichten):

    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.

Adressierung (3)

- Beispiel: m:1



6.2.3 Pufferung

- Def. Kapazität
  - Kapazität eines Kanals bezeichnet 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.

Pufferung (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Keine Pufferung (Kapazität Null)
  - Sender wird blockiert, wenn SEND-Operation vor entsprechender 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.
  - Beispiel: Hoare: Communicating Sequential Processes (CSP).
  - Beispiel: Kommunikation zwischen ADA-Tasks.
  - Synchroner Nachrichtenaustausch wird häufig als zu inflexibel angesehen.

Pufferung (3)

- Beschänkte Kapazität
  - 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 Sendeprozess fährt fort.
  - Ist der Warteraum voll (er enthält N gesendete aber noch nicht empfangene Nachrichten), so wird der Sender blockiert, bis ein freier Warteplatz vorhanden ist.
  - Ablegen der Nachricht kann durch Kopieren der Nachricht oder Speichern eines Zeigers auf die Nachricht realisiert sein.
  - Analog wird Empfänger bei Ausführung einer RECEIVE-Operation blockiert, wenn Warteraum leer.

Pufferung (4)

- Unbeschänkte Kapazität
  - Kanal kann potentiell eine unbeschränkte Anzahl von Nachrichten enthalten.
  - SEND-Operation kann nicht blockieren.
  - Lediglich Empfänger kann bei Ausführung einer RECEIVE-Operation blockieren, wenn Warteraum leer.
  - Implementierung kann erfolgen, in dem Sender und Empfänger die Warteplätze beim Aufruf "mitbringen".

Pufferung (5)

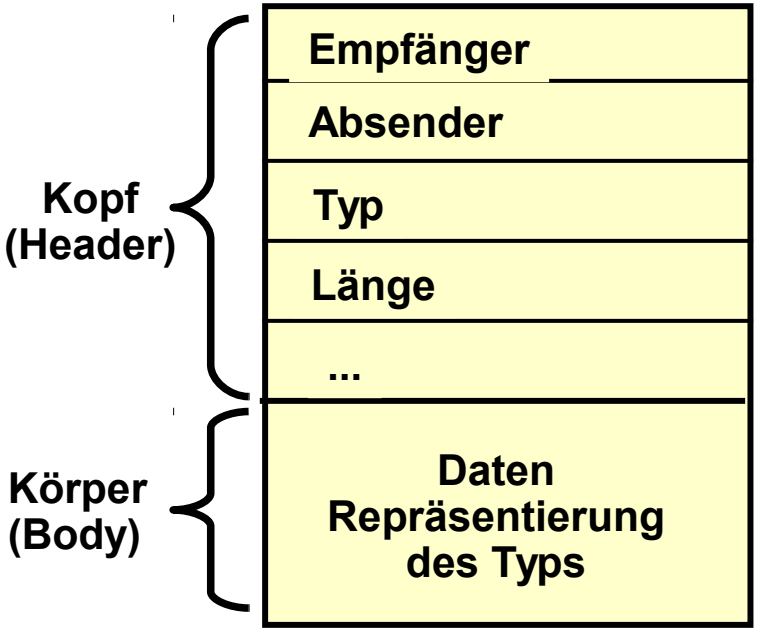
- Konsequenzen
  - Gepufferte Kommunikation (beschränkt oder unbeschränkt) bewirkt zeitlich lose Kopplung der Kommunikationspartner.
  - Nach Abschluss der SEND-Operation weiß der Sender nicht, dass der Empfänger die Nachricht erhalten hat. Er kennt i.d.R. auch keine maximale Zeitdauer dafür.
  - Wenn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

Prozess P (Sender):		**Prozess Q:**
...		**...**
send (Q, *message*);	→	**receive (P, *message*);**
receive (Q, *reply*);	←	**send (P, "acknowledge");**
...		**...**

6.2.4 Semantik von Nachrichten

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Typisierte Nachrichten
  - Nachrichten haben eine typisierte Struktur.
  - Typ ist Sender und Empfänger und z.T. dem Nachrichten-system 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.

6.2.5 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.
  - Fazit: 2-maliges Kopieren. Aufwendig!
- 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).

6.2.6 Beispiel: UNIX

UNIX stellt mehrere Dienste zum Nachrichtenaustausch zwischen Prozessen zur Verfügung:

- Pipes
  - Ursprünglicher Mechanismus zum unidirektionalen Nachrichten-transport (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:

- Vater legt Pipe an, erzeugt zwei Söhne 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 Vater beide Seiten.
- Analog kann der Vater mit einem Sohn über eine Pipe kommunizieren.

Beispiel: Anonyme Pipes (2)

Systemaufrufe:

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

  Erzeugen einer anonymen geöffneten Pipe. Der File-Deskriptor pdfs[0] kann zum Lesen, pdfs[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)`

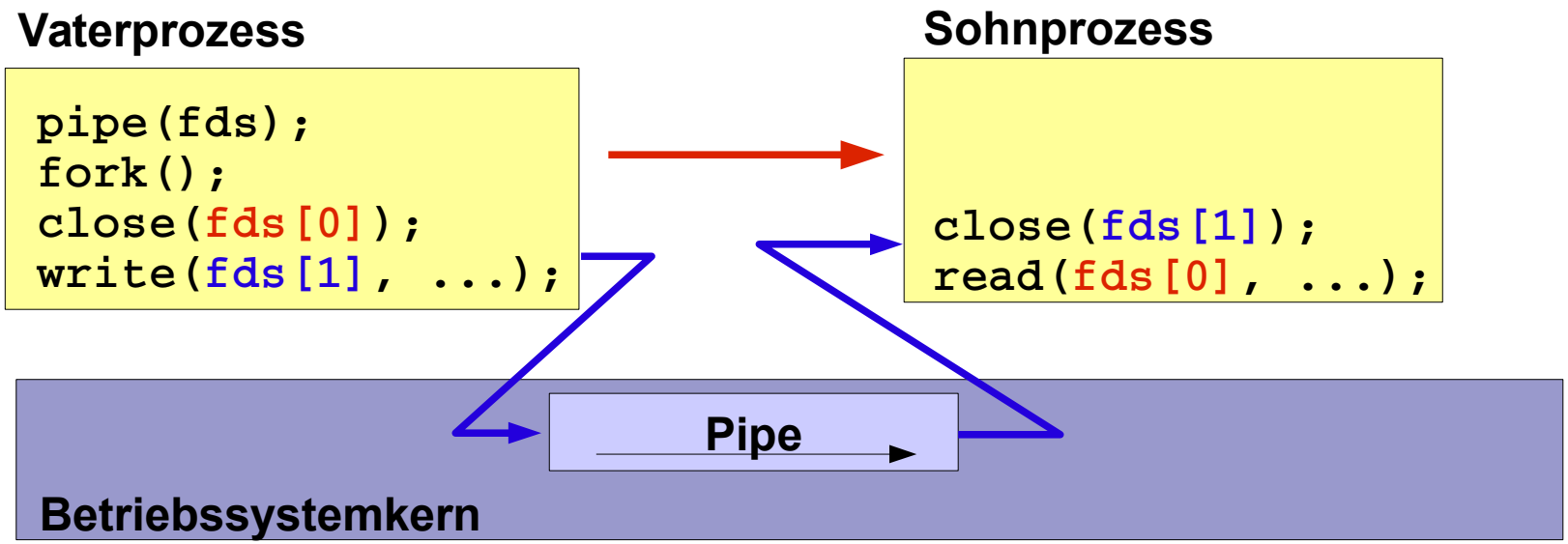
  Empfangen von maximal length bytes aus der durch fd bezeich-neten pipe. Aufrufer blockiert, falls Pipe leer ist. (Blockierung kann analog zu write vermieden werden). Rückgabewert ist die tatsächlich gelesene Anzahl bytes, 0 bei Schließen der Pipe durch die Sender-Seite (Dateiende), -1 bei Fehler.

- `int close(int fd)`

  Schließen eines Endes der Pipe zum Beenden der Kommunikation.

Beispiel: Anonyme Pipes (3)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



- Systemfunktion `int pipe(int fds[2]);`
  erzeugt *zwei* File-Deskriptoren im übergebenen Vektor `fds`:
  `fds[0]` ist zum Lesen geöffnet
  `fds[1]` zum Schreiben
- Rückgabewert: 0 für ok, -1 für Fehler

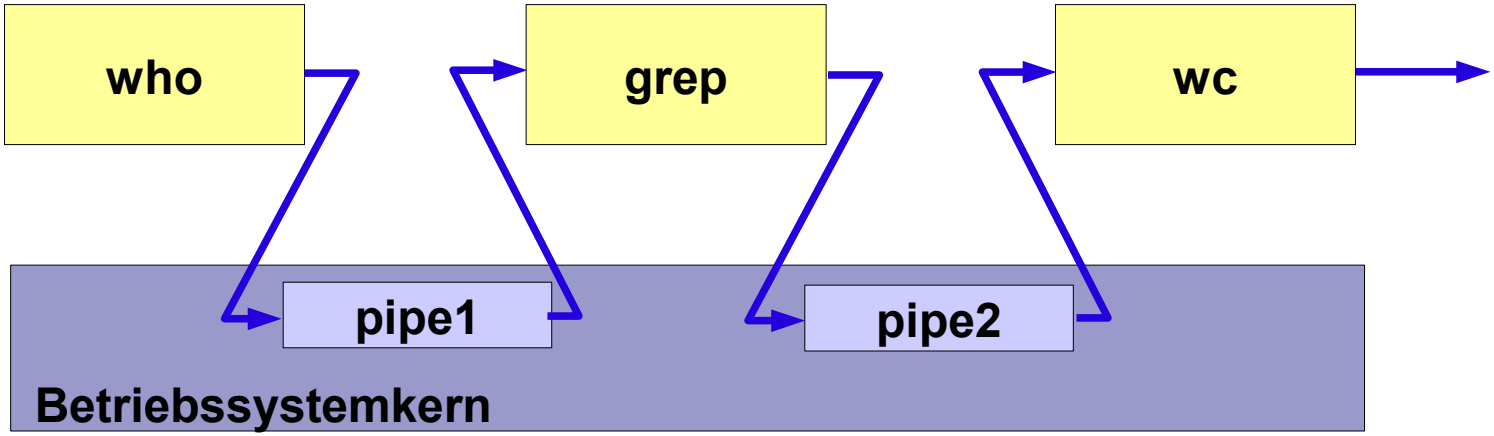
Beispiel: Pipes in der Shell

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Shell-Kommandozeile (wie oft ist „hansl“ auf dem Rechner angemeldet?)

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

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

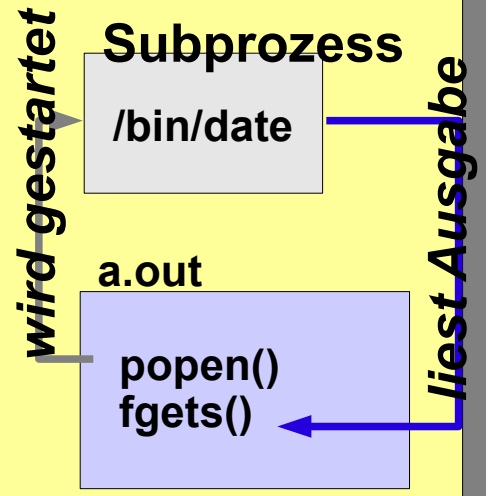


popen()

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

```
#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;
}
```

```
$ ./a.out
Ausgabe von 'date' ist: Di 26. Nov 17:41:19 CET 2013
```



- Mit `popen()` kann ein Kommando als Subprozess gestartet, in dessen Standardeingabe geschrieben (`"w"`) oder dessen Standardausgabe gelesen (`"r"`) werden kann (entweder/oder)
- Verwendung mit „Stream“-Funktionen (`fprintf()`, `fgets()`, ...)
- Schließen mit `pclose()`

Beispiel: 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).

Beispiel: Named Pipes oder FIFOs (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Systemaufrufe:

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

  Mit mknod können allgemeiner 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 V.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.

Vertiefung erfolgt im Praktikum.

Beispiel: Named Pipes oder FIFOs (3)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim


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

int main(void) {
  int fd;
  mkfifo("my_fifo", 0666);
  fd = open("my_fifo", O_WRONLY);
  write(fd, ...);
  ...
  /* Schließen des FIFO */
  unlink("my_fifo");
}
```

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

int main(void) {
  int fd;
  ...
  fd = open("my_fifo", O_RDONLY);
  read(fd, ...);
  ...
}
```

(Fehlerbehandlung weggelassen)

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

Beispiel: Message Queues

Eigenschaften:

- Message Queues (Nachrichtenwarteschlangen) im Rahmen der UNIX System V IPC-Mechanismen eingeführt, Bedeutung gesunken.
- Message Queue definiert Kommunikationskanal, über den mehrere Sender- und mehrere Empfänger-Prozesse, die nicht miteinander verwandt sein müssen, typisierte Nachrichten variabler Länge kommunizieren können.
- Message Queues besitzen eindeutigen Bezeichner aus dem Key-Namensraum, geöffnete Message Queue wird intern über Integer-Id bezeichnet.
- Nachricht besteht aus einem Integer-Nachrichtentyp und variabel langem Nachrichteninhalt. Festlegung der Bedeutung von Typ und Inhalt ist den Prozessen vorbehalten.
- Message Queue hält Berechtigungen zur Zugriffskontrolle.
- Anzahl und Puffergröße der Message Queues werden bei der Systemgenerierung festgelegt.

Beispiel: Message Queues (2)

Systemaufrufe:

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

  Erzeugen einer neuen oder Öffnen einer existíerenden Message Queue, liefert den internen Id der Message Queue.

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

  Senden einer Nachricht an die angegebene Message Queue.

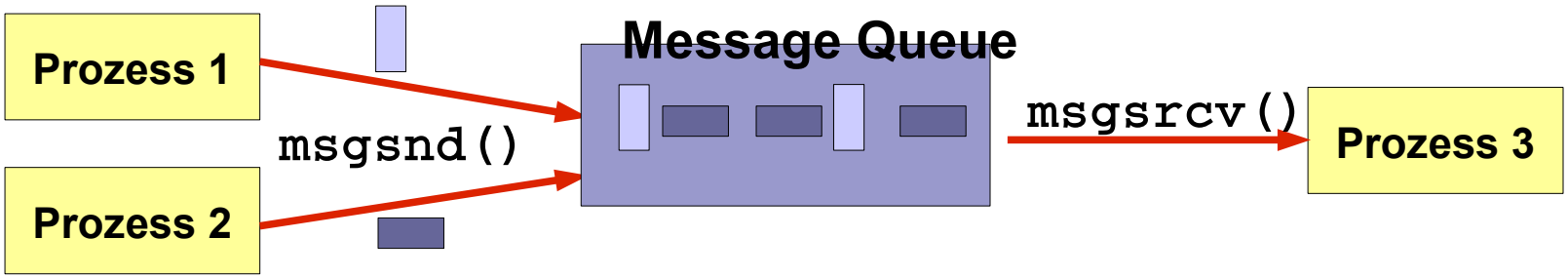
- `int msgrcv(int id, struct msgbuf * buf, int msgsz, long msgtype, int msgflag)`

  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.

Message Queues



- Eine Message Queue ist eine **verkettete** (Nachrichten-) **Liste**, die vom Kernel verwaltet wird
- Empfangsreihenfolge normalerweise „first-in-first-out“, eine Priorisierung der Nachrichten ist aber auch möglich

6.2.7 Beispiel: Windows NT

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Windows NT stellt ebenfalls mehrere Dienste zum Nachrichtenaustausch zwischen Prozessen zur Verfügung:

- Anonyme Pipes

  Semantik ähnlich wie UNIX anonyme Pipes.

- Named Pipes

  geht über die UNIX-Semantik teilweise hinaus: bidirektionaler Kommunikationskanal für mehrere Sende-Prozesse/Threads und einen Empfänger (Server-Prozess), auch auf verschiedenen Windows-Rechnern im Netzwerk.

- Mailslots

  Für jeden Mailslot gibt es einen Empfänger (den Erzeuger des Mailslots) und mehrere mögliche Sender. Wird derselbe Mailslot-Name für mehrere Mailslots auf verschiedenen Rechnern verwendet, so werden gesendete Nachrichten an allen diesen Mailslots zugestellt.

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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.