HOCHSCHULE FULDA
UNIVERSITY OF APPLIED SCIENCES

Betriebssysteme

Stage 6 – Prozesskommunikation

Inter Process Communication (IPC)

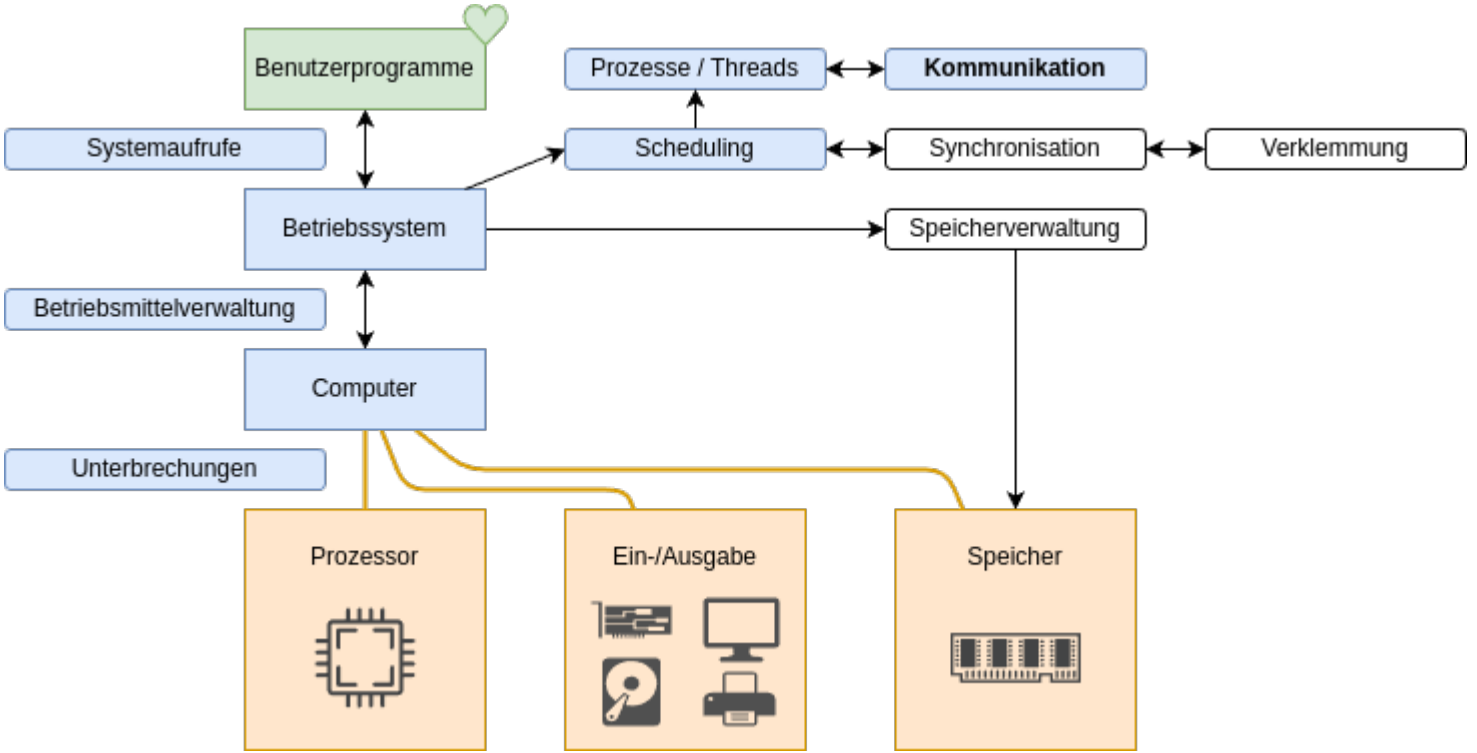
Was bisher geschah ...

- Geschichte der Rechner und Betriebssysteme
- Von-Neumann-Architektur
- Bootprozess und Sicherheitskonzept
- Interaktion mit dem Betriebssystem
  - Systemaufrufe (*system calls*)
  - Unterbrechungen (*interrupts*)
- Aufbau von Betriebssystemen
- Betriebssystemstrukturen

- Prozesse & Threads

- Scheduling
  - Grundlagen
  - Schedulingverfahren

Übersicht


Benutzerprogramme
Systemaufrufe
Betriebssystem
Betriebsmittelverwaltung
Computer
Unterbrechungen
Prozesse / Threads
Kommunikation
Scheduling
Synchronisation
Verklemmung
Speicherverwaltung
Prozessor
Ein-/Ausgabe
Speicher

Interprozesskommunikation (IPC)

- Problemstellung
  - Mehrere Prozesse bearbeiten gemeinsam eine Aufgabe
    - Verkürzung der Bearbeitungszeit durch Nutzung mehrerer CPUs
    - Verbergen der Bearbeitung durch Ausführung im Hintergrund
  - Die Bearbeitung muss koordiniert werden
    - Austausch von Daten zwischen Prozessen
    - Geregelter Zugriff auf gemeinsame Datenstrukturen
    - Reihenfolge der Bearbeitung kann eine Rolle spielen
- Mechanismen zur Kommunikation und Synchronisation notwendig
  - Kommunikation durch gemeinsamen Speicher
  - Kommunikation durch Nachrichtenaustausch
    - Spezialfall: Kommunikation von Ereignissen
  - Synchronisation des Prozessablaufs

Interprozesskommunikation

- Gemeinsamer Speicher (*shared memory*)
  - Memory mapping
- Ereigniskommunikation
  - Grundlagen
  - Beispiel: UNIX Signale
- Nachrichtenaustausch
  - Grundlagen
  - Lokal
    - Beispiel: Pipes
    - Beispiel: Named Pipes
    - *Message queues*
  - Rechnerübergreifend
    - *Beispiel: Sockets / Streams*
- *Entfernte Aufrufe (RPC)*

Shared memory

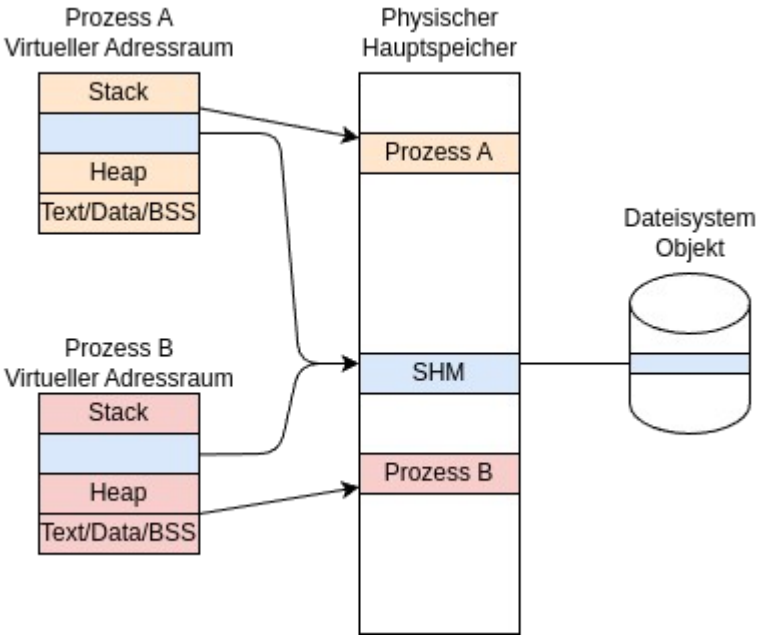
Gemeinsamer Speicher (*shared memory*)

- Gemeinsamer Speicher ermögliche effizienten Datenaustausch zwischen Prozessen

- Realisierung
  - Direkt im Hauptspeicher (System V)
  - Mittels *memory-mapped (special) file* (POSIX, Windows)

- Für den koordinierten Zugriff auf den gemeinsamen Speicher muss bei der Programmierung gesorgt werden

- Hinweis memory-mapping (später bei Hauptspeicher)
  - Memory-mapped files
  - Memory mapped I/O



Schematische Darstellung: Zwei Prozesse mit jeweils eigenem Adressraum und gemeinsamen Speicherbereich

Gemeinsamer Speicher: Implementierungen

- System V (siehe Manpages)
  - Nutzt keys / identifiers zur Referenzierung
  - Festgelegte Größe zum Erstellungszeitpunkt
  - ftok(), shmget(), shmat(), shmdt(), shmctl()
- POSIX (siehe Manpages)
  - Nutzt dateinamen (file descriptors)
  - Veränderung der Größe zur Laufzeit möglich
  - shm_open(), mmap(), shm_unlink()
- Linux
  - Implementiert die zuvor genannten Standards und eine Reihe eigener Erweiterungen (z.B: MAP_ANONYMOUS)
- WindowsNT (Dokumentation)
  - CreateFileMapping(), OpenFileMapping()
  - MapViewOfFile(), UnmapViewOfFile()
  - CloseHandle(), DeleteFile()

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>

void * get_shm (size_t size) {
    /* Protection determines whether read, write, execute,
     * or a combination of these access permissions are
     * applied to the memory segment being mapped.
     */
    int protection = PROT_READ | PROT_WRITE;

    /* The flags argument determines whether updates to
     * the mapping are visible to other processes mapping
     * the same region, and whether updates are carried
     * through to the underlying file.
     */
    int flags = MAP_SHARED | MAP_ANONYMOUS;

    /* Create the shared memory segment.
     * Due to MAP_ANONYMOUS, we can just pass -1 and 0 as
     * the last two arguments, which would otherwise be a
     * file descriptor and size.
     */
    void* m = mmap (NULL, size, protection, flags, -1, 0);
}
```

Ereigniskommunikation

Ereigniskommunikation

- Mechanismus für die Kommunikation (Signalisierung) von Ereignissen
  - Stellt einen Spezialfall der Nachrichten-orienterten Kommunikation
  - Kann auch zur Synchronisation eingesetzt werden

- Ereignisse zeigen das Eintreten einer "wichtigen Begebenheit" an
  - das Erreichen eines bestimmten Zustands
  - den Ablauf einer vorgegebenen Zeitspanne
  - das Eintreten eines Fehlers
  - usw.

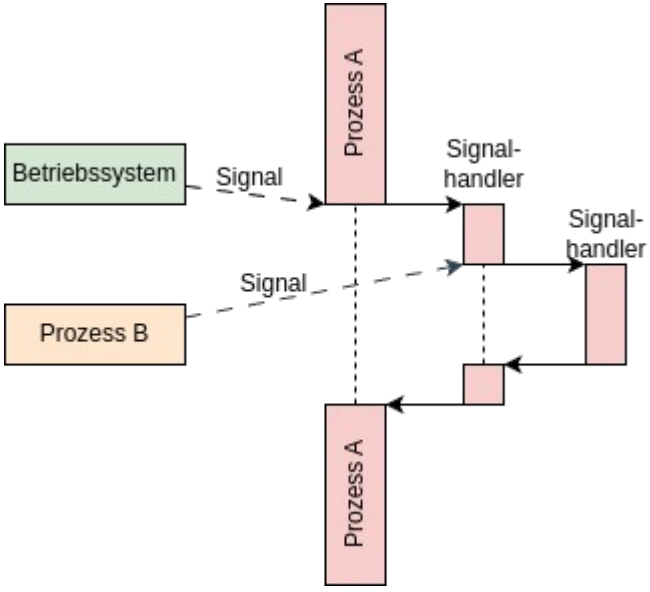
- Bereitstellung eines effizienten Ereigniskonzepts ist für Echtzeitbetriebssysteme von großer Bedeutung
  → In Echtzeitanwendungen modellieren Ereignisse häufig reale Ereignisse in der Außenwelt

Ereigniskommunikation

- Das UNIX Signalkonzept überträgt das Hardware-Interrupt-Konzepts auf die Software-Ebene
  - Interrupt → Signal
  - Interrupt Handler → Signal Handler
  - Maskieren von Interrupts → Maskieren von Signalen

- Ereignisse werden lediglich nummeriert
  - Die Zuordnung einer tatsächlichen Bedeutung kann ganz oder teilweise vorgegeben oder den Anwendungen überlassen sein

- UNIX unterscheidet zwei Arten von Signalen
  - Fehler-Ereignisse, die bei der Programmausführung auftreten
    - SIGSEGV → Adressierung eines ungültigen Speicherbereichs
    - SIGILL → Versuch der Ausführung einer ungültigen Instruktion
  - Asynchrone Ereignisse, die von außerhalb des Prozesses stammen
    - SIGCHLD → Beendigung eines Kind-Prozesses
    - SIGHUP → Terminalleitungstreiber zeigt Ende der Übertragung an
    - SIGALRM → Programmierter Timer läuft ab



Funktionsweise Signale

- Versenden von Signalen
  - Ein Prozess im Zustand *rechnend* kann durch den BS-Kern Signale senden und empfangen
  - 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 andere Prozesse senden.
  - Der BS-Kern kann jedes Signal an jeden Prozess senden

- Empfang von Signalen
  - *Rechnende* und *nicht rechnende* Prozesse können Signale von rechnenden Prozessen empfangen
  - Ein Prozess kann festlegen, dass er bestimmte Signale selbst durch einen Signal-Handler behandeln oder ignorieren will. Einige Signale (z.B. SIGKILL) können nicht abgefangen werden.

- Verarbeitung anstehender Signale geschieht beim Übergang vom Kern- in den Benutzermodus
  - 1) Zielprozess im Zustand *running*
    Unmittelbare Unterbrechung und Start der Signal-Behandlungsroutine
  - 2) Zielprozess im Zustand *ready*
    Das Betriebssystem vermerkt das Signal im PCB. Bearbeitung sobald der Prozess die CPU erhält
  - 3) Zielprozess im Zustand *blocked*
    Die Blockierung wird unterbrochen (EINTR), der Prozesszustand auf ready gesetzt, danach wie 2

System V Signale

- **`void (*signal(int sig, void (*func)(int)))(int);`**
  Installieren eines Signalhandlers.
- **`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 = 0 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).
- **`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 SIG_IGN gesetzt und damit bis auf weiteres vom aufrufenden Prozess ignoriert
- **`void alarm (int seconds);`**
  Ein Prozess kann darum bitten, dass ihm nach Ablauf von seconds das Signal SIG_ALRM zugestellt wird

POSIX Signale

- Moderne und kompatible Lösung
  - Zuordnung einer Aktion bei Auftreten des Signals `sig` und Speichern der bisherigen Aktion
  - Entspricht POSIX (aber auch BSD `sigvec()` oder System V `signal()`)
  - Erfolgt nur einmal im Programm für jedes Signal, für das die Default-Aktion geändert werden soll
  - Fehler `EINVAL`, falls Signalhandler nicht überschreibbar (`sig == SIGKILL` oder `sig == SIGSTOP`)
  - Nach Abschluss der Behandlung → Fortsetzung des Programms an der unterbrochenen Stelle

- Beispiel

```
#include <signal.h>

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


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

Nachrichtenaustausch

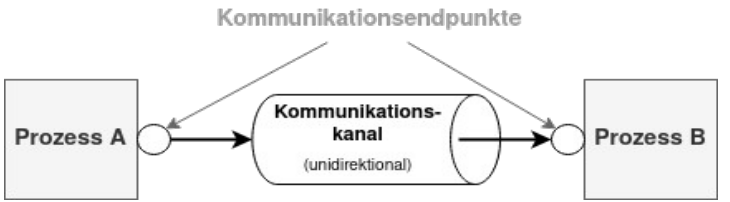
Nachrichtenbasierte Kommunikation

- Kopplung von Kommunikationspartnern, um die Kommunikation zu ermöglichen
  - Die Verbindung heißt Kommunikationskanal oder Kanal
  - Ein Prozess kann zu einem Zeitpunkt mehrere Kanäle aufrecht erhalten (z.B. zu versch. Prozessen)
  - Kanal heißt gerichtet oder unidirektional, wenn ein Prozess ausschließlich die Sender-Rolle, der andere ausschließlich die Empfänger-Rolle ausübt, ansonsten ungerichtet oder bidirektional

- Primitive
  - **SEND (destination, message)**
    Senden einer Nachricht durch einen Sendeprozess oder einfach Sender

  - **RECEIVE (source, message)**
    Empfangen einer Nachricht durch einen Empfangsprozess oder Empfänger

- Wichtige Entwurfsaspekte eines Nachrichtensystems
  - Adressierung
  - Pufferung
  - Nachrichtenstruktur



Adressierung bei Nachrichtenaustausch

- Unterscheidung: Anzahl der Kommunikationsteilnehmer eines Kanals
  - Genau zwei → Direkte bzw. Indirekte Adressierung (Unicast)
  - Mehr als zwei → Gruppenkommunikation (Multicast, Broadcast)

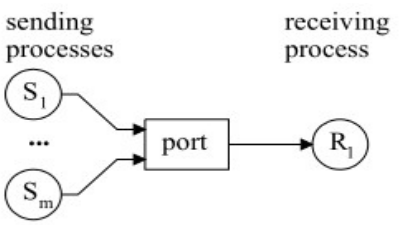
- Direkte Adressierung
  - Prozesse haben eindeutige Adressen

  - Symmetrische Variante
    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

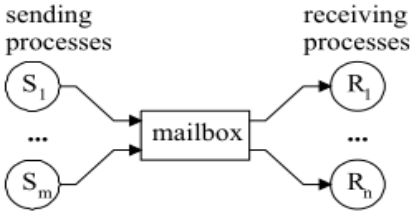
  - Asymmetrische Variante
    Sendender Prozess muss Empfänger benennen, diesem wird mit Empfang die Identität des Senders bekannt
    - SEND (P, message) // Sende eine Nachricht an Prozess P
    - RECEIVE (sender_id, message) // Empfange eine Nachricht. sender_id bezeichnet den Absender

Adressierung bei Nachrichtenaustausch

- 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
  - Vorteil
    - 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



M:1 Beziehung



M:N Beziehung

Synchronisation bei Nachrichtenaustausch

- Synchrone Kommunikation
  - Empfänger blockiert bis die Nachricht eingetroffen ist
  - Sender blockiert bis der Empfang der Nachricht bestätigt ist

- Asynchrone Kommunikation
  - Sender übergibt die Nachricht dem Betriebssystem und arbeitet weiter
  - Blockierung (warten) ist beim Empfänger und Sender optional
  - Pufferung von Nachrichten erforderlich

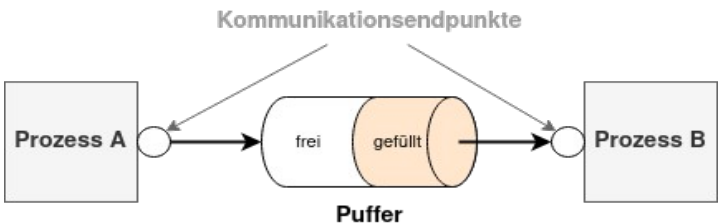
- In der Praxis
  - Häufig asynchroner Nachrichtenaustausch mit Blockierung (wo nötig)
  - Zum Beispiel: nicht blockierendes Senden und blockierendes Empfangen

Pufferung

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

- Implementierung
  - Die Pufferungsfähigkeit eines Kanals wird i.d.R. durch einen Warteraum/Warteschlange im Betriebssystemkern erreicht

- Grenzwertbetrachtung
  - Keine Pufferung
  - Beschränkte Kapazität
  - Unbeschränkte Kapazität



Pufferung / Kapazität

- Keine Pufferung (Kapazität Null)
  - Kanal kann keine Nachrichten enthalten, sondern Sender und Empfänger müssen sich “treffen” (Rendezvous) und die Nachrichten direkt verarbeiten

    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

- Beschränkte Kapazität
  - Kanal kann zu einem Zeitpunkt maximal N Nachrichten enthalten (Normalfall)

    Im Falle einer SEND-Operation bei nicht-vollem Warteraum wird die Nachricht im Warteraum abgelegt und der Sendeprozess fährt ohne Blockierung 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. Analog wird der Empfänger bei Ausführung einer RECEIVE-Operation blockiert, falls der Warteraum leer

- Unbeschränkte Kapazität
  - Der Kanal kann potentiell eine unbeschränkte Anzahl von Nachrichten enthalten

    Die SEND-Operation kann nicht blockieren. Lediglich der Empfänger kann bei der Ausführung einer RECEIVE-Operation blockieren, falls der Warteraum leer ist

Pufferung / Kapazität

- Gepufferte Kommunikation bewirkt zeitlich lose Kopplung der Kommunikationspartner
  - Kommunikationspartner müssen nicht aufeinander warten (kein “Rendevouz” nötig)

- Konsequenz
  - Nach Abschluss der SEND-Operation weiß der Sender nicht, wann genau der Empfänger die Nachricht erhält bzw. ob er diese bereits erhalten hat
  - Er kennt i.d.R. auch keine maximale Zeitdauer dafür

- Lösung
  - Wenn das Wissen über die Ankunft einer Nachricht beim Empfänger wesentlich für den Sender ist, muss dafür eine explizite Kommunikation zwischen Sender und Empfänger durchgeführt werden

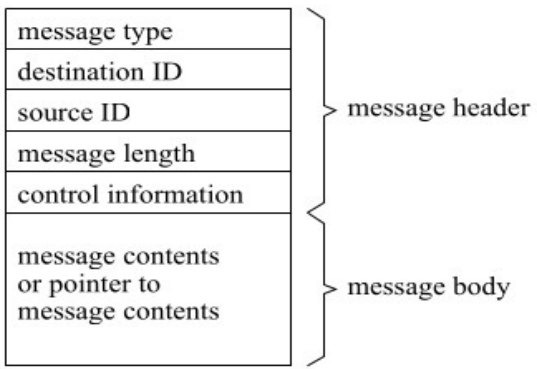
Nachrichtenformat

- Damit Kommunikationspartner sich gegenseitig „verstehen“ können, müssen diese sich auf ein gemeinsames Nachrichtenformat verständigen

- Typisierte Nachrichten
  - Nachrichten haben in der Regel eine typisierte Struktur
  - Der Typ ist Sender und Empfänger (und z.T. dem Nachrichtensystem) bekannt und wird in Operationen verwendet

- 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

- Stromorientiert (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



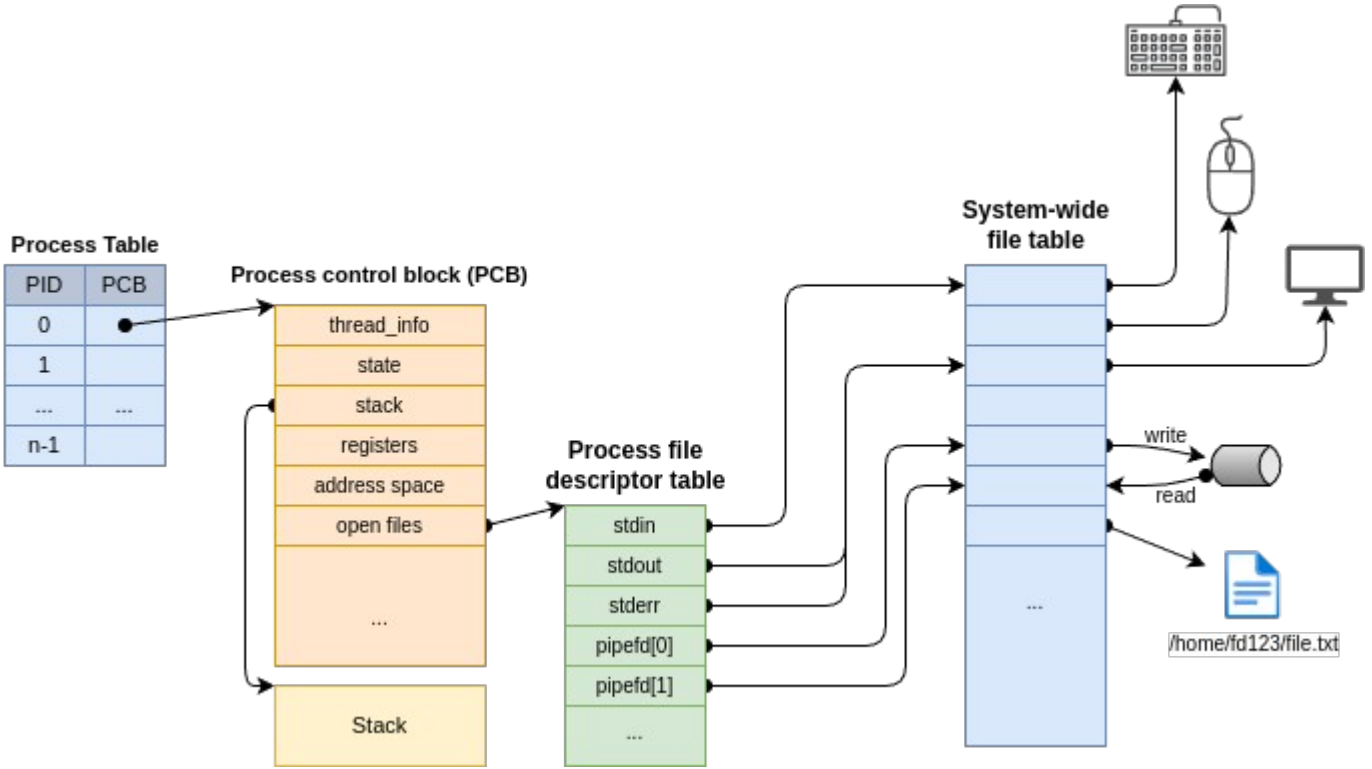
Beispiele

Übersicht: UNIX Nachrichtenaustausch

- (Anonyme) Pipes
  - Unidirektionaler Nachrichtentransport zwischen verwandten Prozessen
  - Vererbung der Kommunikationsendpunkte durch fork()
- Named Pipes (oder FIFOs)
  - Erweiterung auf nicht-verwandte Prozesse durch Repräsentierung im Dateisystem
- Message Queues
  - Message Queue ist ein komplexes, gemeinsam genutztes Objekt zum Austausch typisierter Nachrichten zwischen potentiell beliebig vielen Sende- und Empfangs-Prozessen (Bestandteil der System V IPC-Mechanismen)
- Sockets
  - In 4.2BSD UNIX zusammen mit TCP/IP eingeführtes Konzept zur allgemeinen, rechnerübergreifenden, bidirektionalen, nachrichtenorientierten Interprozesskommunikation
- STREAMS
  - Mit UNIX System V Rel. 3 eingeführte Gesamtumgebung zur Entwicklung von geschichteten IPC-Protokollen

Pipes

- Eigenschaften
  - Unidirektional
  - Gepuffert
  - Zuverlässig
  - Stromorientiert
- Varianten
  - Anonyme Pipes
  - Named Pipes (FIFOs)



File descriptors eines Prozesses (u.a. auch Pipes)
Stark vereinfachte Darstellung

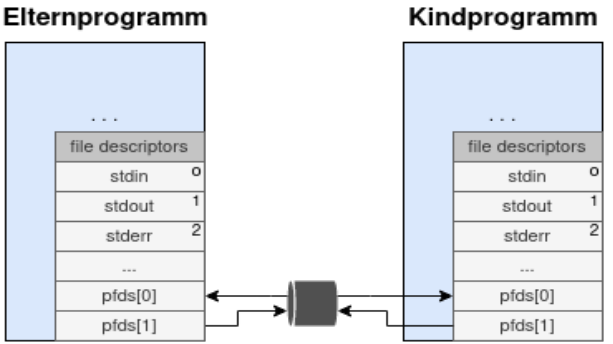
Anonyme Pipes

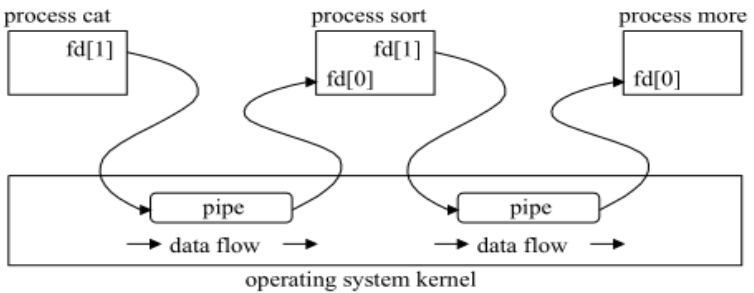
- Eigenschaften
  - Erlaubt zwei verwandten Prozessen (Eltern-Kind / Kind-Kind) einen Bytestrom zu kommunizieren
  - Fixe Kapazität von einigen Speicher-Seiten
    (z.B. Linux: 16 Seiten = 64 kB)
  - Wird von den Prozessen wie eine Datei behandelt/benutzt

- Typische Verwendung
  - Elternprozess legt Pipe an
  - Aufruf von fork(), wodurch die File-Deskriptoren (die geöffneten Enden der Pipe) vererbt werden
  - Der Schreiber schließt die Lese-Seite
  - Der Leser schließt die Schreib-Seite

  Analog können auch zwei Kindprozesse über die Pipe kommunizieren. In dem Fall schließt der Elternprozess beide Seiten der Pipe.

  Werden die Enden der Pipe als stdin bzw. stdout verwendet, so wird dadurch die aus der Shell bekannte Pipe-Funktionalität erreicht. (→ man dup2)





Anonyme Pipes

- **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. Der Aufrufer blockiert, falls die 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. Der Aufrufer blockiert, falls die Pipe leer ist (Blockierung kann analog zu write vermieden werden). Rückgabewert ist die tatsächlich gelesene Anzahl Bytes, oder 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

Benannte Pipes

- Benannte Pipe (*Named Pipe*, auch FIFO)
  - Kommunikationskanal über den mehrere Sender- und mehrere Empfänger-Prozesse einen gemeinsamen Bytestrom kommunizieren können
  - Prozesse müssen nicht miteinander verwandt sein
  - besitzt einen Namen aus dem Dateinamensraum und eine Repräsentierung im Dateisystem aber keine Datenblöcke, sondern lediglich einen Seiten-Puffer im Arbeitsspeicher
  - besitzt Zugriffsrechte wie bei Dateien, die beim Öffnen überprüft werden

- In System V sind FIFOs durch ein spezielles Dateisystem (fifofs) implementiert

Named Pipes

- **int mknod (char* pfad, int modus, int dev);**
  - Mit mknod können beliebige “special files” im Dateisystem angelegt werden – auch FIFOs.
    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 System V v.4 / Linux existiert zusätzlich mkfifo()).

- **int mkfifo (const char *pathname, mode_t mode);**
  - Erzeugen einer Named Pipe (FIFO) im Dateisystem. Der Parameter pathname gibt den absoluten Namen des FIFO im Dateisystem an, mit mode können die Zugrifsrechte gesteuert werden.

- **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.

- **int write (int fd, char* buf, unsigned length);**
  - Senden von length bytes in die durch fd bezeichnete Pipe (vgl. Anonyme Pipe).

- **int read (int fd, char* buf, unsigned length);**
  - Empfangen von maximal length bytes aus der durch fd bezeichneten Pipe (vgl. Anonyme Pipe).

- **int close (int fd);**
  - Schließen eines Endes der Pipe zum Beenden der Kommunikation (vgl. Anonyme Pipe).

Message Queues

- Message Queues (Nachrichtenwarteschlangen)
  - Wurden im Rahmen der UNIX System V IPC-Mechanismen eingeführt
  - Definiert einen Kommunikationskanal, über den mehrere Sender- und mehrere Empfänger-Prozesse, die nicht miteinander verwandt sein müssen, typisierte Nachrichten variabler Länge kommunizieren können
    → Ungerichtete M:N-Kommunikation

- Eigenschaften
  - Adressierung durch rechnerweit eindeutige Adressen (Key-Namensraum)
  - Nachricht besteht aus einem Integer-Nachrichtentyp (long) und variabel langem Nachrichteninhalt (Festlegung der Bedeutung von Typ und Inhalt ist den Prozessen vorbehalten)
  - Hält Berechtigungen zur Zugriffskontrolle
  - Anzahl und Puffergröße der Message Queues werden bei der Systemgenerierung festgelegt
  - Sende- und Empfangsoperationen können festlegen ob
    - Blockierend bzw. nicht blockierend
    - Empfang aller Nachrichten oder nur solche eines bestimmten Typ

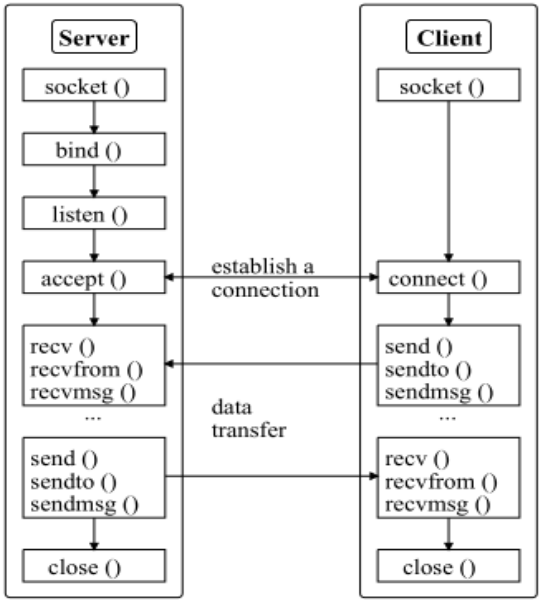
- Message Queues haben an Bedeutung verloren
  → Nutzung von Sockets erlaubt lokale und auch rechnerübergreifende Kommunikation

Siehe: Windows Mailslots

Sockets

- Sockets bieten Endpunkte für Kommunikationsverbindungen
  - jedem Socket ist genau ein Prozess zugeordnet
- Sockets beziehen sich auf eine bestimmte Domäne
  - 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)
- Beispiele für Domänen
  - UNIX Domäne
  - Internet Domäne
  - Apple Talk, u.v.m.
- Internet Domäne wird in Verteilte Systeme behandelt
  - Hier wäre struct sockaddr_in in Verwendung

```
sin_family  = AF_INET6 / AF_INET
sin_port    = 16-Bit Portnummer
sin_addr    = IPv4-Adresse
```




```c
#include <sys/socket.h>

// Create a socket
int socket (int domain, int type, int protocol);

// Bind to a specific address
int bind (int socket,
          const struct sockaddr *address,
          socklen_t address_len);
```

Sockets: UNIX Domain Socket

- Lokaler, bidirektionaler, zuverlässiger Kommunikationskanal
  - Im POSIX-Standard beschriebene Methode zur Interprozesskommunikation auf UNIXoiden Systemen
  - Besitzt einen Namen aus dem Dateinamensraum und eine Repräsentation im Dateisystem aber keine Datenblöcke, sondern lediglich einen Seiten-Puffer im Arbeitsspeicher
  - Dateiberechtigungen erlauben die Zugriffssteuerung

- Kenntnis über die Lokalität des Sockets
  - Sender schreibt direkt in den Empfangs-Puffer des Empfängers
  - Keine Notwendigkeit für das Anfügen von Headern, die Berechnung von Checksummen, usw.

```
1   #include <sys/socket.h>
2   #include <sys/un.h>
3
4   // Socket address datastructure
5   struct sockaddr_un sock;
6   bzero ((char *) &sock, sizeof (...));
7   sock.sun_family = AF_UNIX;
8   strcpy (sock.sun_path, "/tmp/mysocket");
9
10  // Create a UNIX domain socket
11  fd = socket (PF_UNIX, SOCK_DGRAM, 0)
12
13  // Bind the socket to the device file
14  unlink ("/tmp/mysocket");
15  bind (fd, &sock, sizeof (...))
16
17  // Read from the socket
18  len = recvfrom (fd, buffer, sizeof (...), 0,
19                  &sock, sizeof (...));
```

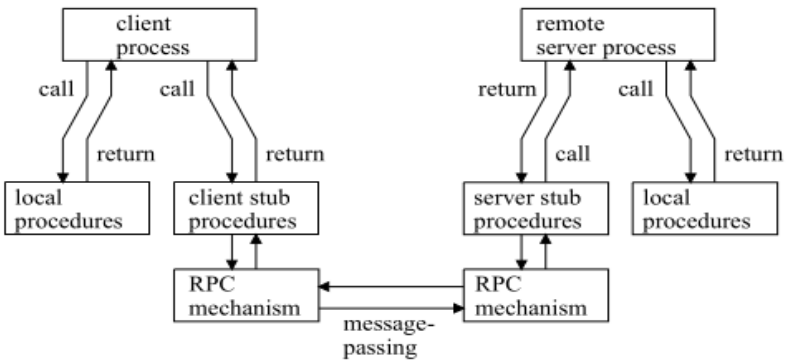
Remote Procedure Call (RPC) (Tanenbaum Kapitel 8.2.4)

- Ein Prozess erhält eine „Dienstleistung“ eines entfernten Prozesses durch die Verwendung eines „gewöhnlichen“ Prozeduraufruf
  - Ursprüngliche Entwicklung durch Sun Microsystems, aber auch Teil des Xerox Network Systems (XNS)
  - Netzwerk und Kommunikationsmechanismen sind transparent für das Anwendungsprogramm
  - Geringe Fehleranfälligkeit (Sende- und Empfangsoperationen, sowie Datenkonvertierungen automatisch)

- Funktionsweise
  - RPC-Schnittstelle wird in spezieller Sprache beschrieben
    (Interface Definition Language (IDL), Microsoft Interface Definition Language (MIDL), RPC Language (RPCL))
  - spezieller Compiler erzeugt aus der IDL-Beschreibung u. a. die Platzhalter-Prozeduren (*stubs*), die lokale Prozeduraufrufe in RPC-Prozeduraufrufe überführen
    (Unix: rpcgen, Microsoft Windows: MIDL Compiler)

- IDL-Beschreibungen verschiedener Systeme sind im Allgemeinen nicht kompatibel

- Beispiel: Linux D-BUS, Java RMI, Sun NFS, SOAP (Simple Object Access Protocol)



Zusammenfassung

- Nachrichtenbasierte Kommunikation
  - Adressierungsart
  - Pufferung
  - Nachrichtenformat
  - Arten: (Signale), Pipes, Message Queues, Sockets
- Ereigniskommunikation
  - Signale (als Spezialfall von Nachrichtenaustausch)
- Shared memory