Soft- und Hardware-Systeme

speziell

Verteilte Systeme

Fachhochschule Niederrhein

SS 2006

Prof. Dr. Rethmann
Prof. Dr. Ueberholz

Inhalt

Linux → Plattform für Übungen/Praktikum

- Aufgaben eines Betriebssystems
- Einführung in Linux/Unix
- Prozesse und Threads

Einführung in verteilte Systeme

- Historische Entwicklung
- Versuch einer Definition
- Motivation und Beispiele
- Anforderungen (sicher, skalierbar, fehlertolerant, ...)

Inhalt (2)

Client-Server-Strukturen

- Client-Server
  * Interaktionssemantik: blockierend, nicht blockierend, Anforderung/Antwort geht verloren, ...
  * Serveraktivierung: per request, process per service, ...
  * Serverzustände: zustandsinvariant, -ändernd
  * Caching: Konsistenzproblem
- Client-Server-Server
  * Broker: welcher Server stellt gesuchten Service bereit
  * Trader: aufgrund gewünschter Service-Qualität einen Server aus mehreren auswählen
  * Proxy, Balancer, ...

Inhalt (3)

Programmiermodelle

- Nachrichtenbasiert (sockets)
- Auftragsorientiert (remote procedure call)
- Objektbasiert (Java RMI, Corba)
- Komponentenbasiert (EJB)

Dienste

- Namensdienst (DNS)
- Verzeichnisdienst (NFS, DFS)
- Zeitdienst (NTP)
- Sicherheitsdienst (SSL)

Inhalt (4)

Synchronisierung

- Wechselseitiger Ausschluss
- Transaktionen (commit/rollback, Serialisierbarkeit durch Sperren/Zeitstempel, ...)
- Wahlalgorithmen (Server-Ausfall → neuen Master wählen)

Fehlertoleranz

- Konzepte, Fehlermodelle, Redundanz
- Prozess-Elastizität (mehrere Prozesse in einer Gruppe, Fehlermaskierung durch Replikation von Prozessen und Einigung, Konsensalgorithmen wie 2-aus-3)

Inhalt (5)

Fehlertoleranz (Fortsetzung)

- Zuverlässige Client/Server-Kommunikation
- Zuverlässige Gruppenkommunikation

Cluster-Computing

- Motivation, Architektur
- Klassifikation (shared disk/nothing/everything, homogen/heterogen, high availability/performance, MIMD)
- Programmiermodelle (PVM: Parallel Virtual Machine, MPI: Message Passing Interface, Master/Slave)

Literatur

- Günter Bengel:
  Verteilte Systeme. Vieweg Verlag, 2000
- A. Tanenbaum, M. van Steen:
  Verteilte Systeme. Pearson Studium, 2003
- A. Tanenbaum, J. Goodman
  Computerarchitektur. Pearson Studium, 2001
- A. Tanenbaum
  Moderne Betriebssysteme. Pearson Studium, 2002
- R. Stones, N. Matthew
  Linux Programmierung. MITP-Verlag, 2000

Linux

Aufgaben eines Betriebssystems

Ziel der Systemprogrammierung:

- Anwender vor der Komplexität der Hardware bewahren
- einfache virtuelle Maschine bereitstellen
- Ressourcen gerecht, effizient und sicher verwalten

Konzept der Software-Schichten:

Spiele	Textverarbeitung	Tabellenkalkulation	} Anwendungsprog.
Editor	Compiler	Kommandointerpreter	} Systemprogramme
Betriebssystem			
Maschinensprache			} Hardware
Mikroprogrammierung			
physikalische Geräte			

Aufgaben eines Betriebssystems (2)

Mikroprogramm:

- gewöhnlich im ROM (read only memory) gespeichert
- umsetzen von Maschinenbefehlen (add, move, jump) in eine Folge einzelner Schritte.

Maschinensprache:

- Menge aller Befehle, die Mikroprogramm interpretiert
- Ein- und Ausgabegeräte werden durch Geräteregister (device register) angesprochen
- Beispiel Plattensteuerung: Plattenadresse, Hauptspeicheradresse, Byte-Anzahl, Übertragungsrichtung, ...

Aufgaben eines Betriebssystems (3)

Bei frühen Rechnersystemen wurde Hardware direkt vom Anwenderprogramm angesteuert:

- jeder Programmierer muss Hardware-Zugriff umsetzen
- fehleranfällig, Expertenwissen notwendig
- bindet Programme an die vorhandene Hardware

→ **das BS als erweiterte/abstrakte/virtuelle Maschine**

- Gerätetreiber (device driver) übernehmen den Zugriff auf die Hardware und stellen den Anwenderprogrammen eine einheitliche Schnittstelle bereit
- Vorteil: Programme müssen nicht angepasst werden, wenn die Rechnerkonfiguration geändert wird → nur austauschen des Treibers bei gleicher Schnittstelle

Aufgaben eines Betriebssystems (4)

Prinzip der Hardware-Zugriffe:

- erfolgen durch Port-Zugriffe (Lesen/Schreiben spezieller Speicherzellen regt eine Aktion der Hardware an)
- einzelner Zugriff löst nur wenig Aktion aus
- nützliche Aktion erfordert ganze Reihe von Zugriffen
- Gerätetreiber fasst Zugriffssequenzen zusammen

→ vermindert das Fehlerrisiko

Aufgaben eines Betriebssystems (5)

Schutz des Systems vor fehlerhaften Programmen:

- bei Einbenutzer- (single-user-)Systemen kann der Benutzer nur seine Daten und seine Hardware beschädigen
- Beispiele: MS-DOS und darauf basierende MS-Windows-Versionen (bis 3.11), alte DOS-Spiele umgehen BS

weitere Aufgaben in Mehrbenutzer- (multi-user-)Systemen:

- Zugriffsbeschränkungen: welche Benutzer dürfen auf welche Daten, Geräte, usw. zugreifen
- Zuteilung verfügbarer Geräte: Zugriff auf Drucker gesperrt, wenn belegt (spooling - simultaneous peripheral operation on line). Wechseln einer CD gesperrt, wenn noch Zugriffe. ...

Aufgaben eines Betriebssystems (6)

weitere Aufgaben: (Fortsetzung)

- Virtualisierung: Ressourcen können scheinbar gleichzeitig von mehreren Benutzern angesprochen werden. Beispiele: Hauptspeicher, Prozessor, Festplatte, ...

→ **das BS als Ressourcenverwalter**

Konzept von Unix

geschützter Betrieb:

- direkten Hardware-Zugriff verhindern (alte DOS-Spiele erreichen durch direkten Hardware-Zugriff unter Umgehung des Betriebssystems höhere Performance)
- mit frühen Prozessoren nicht möglich: Programm kann jederzeit alle Befehle verwenden
- später: geschützter Modus (protected mode)
  * im protected mode dürfen Programme keine Befehle ausführen, die auf externe HW zugreifen
  * Betriebssystem muss solche Befehle ausführen können

→ **einführen eines Privilegiensystems**

Konzept von Unix (2)

einfaches Privilegiensystem: Alles oder nichts

- Programme mit/ohne Hardware-Zugriffsrechten
- aufteilen in **Systemkern (Kernel)** und Anwenderprogramme: Kernel übernimmt elementare BS-funktionen, Anwenderprogramm fordert beim Kernel alle benötigten Dienste an (Systemaufrufe, system calls)

Grundfunktionen eines BS können damit erfüllt werden:

- Hardware-Treiber im Kernel sorgen für Abstraktion
- Schutz durch Privilegientrennung
- da alle Ressourcen über den Kernel angesprochen werden, kann er diese zuteilen und virtualisieren

Geschichte der Betriebssysteme

erste Generation: (1945 - 1955) Röhren

- kleine Gruppen von Spezialisten, die Maschinen entwarfen, bauten, programmierten, betrieben und warteten
- Programmierung in Binärcode
- keine Programmiersprachen, kein Assembler
- kein Betriebssystem
- Zeit zur Benutzung der Maschine wurde auf Wandtafeln den Programmierern zugeteilt
- Programme wurden auf Lochkarten gespeichert
- lösen numerischer Problemstellungen, z.B. berechnen von Sinustabellen

Geschichte der Betriebssysteme (2)

zweite Generation: (1955 - 1965) Transistoren

- Computer wurden zuverlässig/konnten verkauft werden
- erstmals klare Trennung zwischen Entwickler, Hersteller, Betreiber, Programmierer und Wartungspersonal
- Maschinen in gesicherten, klimatisierten Räumen, betreut von geschulter Gruppe von Bedienern (operator)
- enorm hohe Investitionskosten, nur große Firmen, Behörden und Universitäten können sich mehrere Millionen Dollar teuer Anlagen leisten
- erste Programmiersprachen (Fortran/Assembler)

Geschichte der Betriebssysteme (3)

Ausführen eines Rechenauftrags (job):

- schreiben des Programms auf Papier
- Lochkarten stanzen
- abgeben der Lochkarten beim Bedienungspersonal im Eingaberaum
- Bedienungspersonal legt ggf. den Lochkartenstapel des Compilers und das Programm in den Kartenleser
- Bediener entnimmt dem Drucker die Ausgabe und trägt sie zum Ausgaberaum

Oft war die Ausgabe der Fehlerbericht des Compilers oder der Speicherinhalt (core dump) des abgestürzten Programms und der Ablauf begann von vorne.

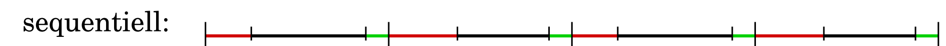
Geschichte der Betriebssysteme (4)

Stapelsystem (batch system) reduziert vergeudete Zeiten.

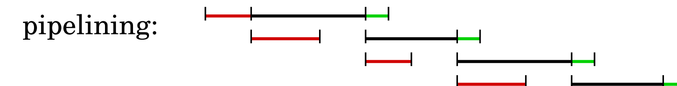
- Einsatz von drei verschiedenen Rechnern:
  1. einlesen der Lochkarten und kopieren auf Band
  2. Band einlesen, rechnen, Ausgabe auf Band schreiben
  3. Band auslesen und Ausgabe drucken
- ausnutzen von Pipelining:

(1) Lochkarten lesen (2) Rechnen (3) Ausgabe drucken

sequentiell:



pipelining:



Geschichte der Betriebssysteme (5)

dritte Generation: (1965 - 1985) ICs

- Mehrprogrammbetrieb durch Prozess-Konzept:
  * Prozess: Programm inklusive Daten, Stack, Befehlszähler usw. Voraussetzung: Hauptspeicheraufteilung
  * bei Datenein-/ausgabe ist CPU untätig und kann auf anderen Prozess umgeschaltet werden

  ⇒ **bessere Ausnutzung der CPU**
- Aufträge direkt auf Platte zwischenspeichern (spooling): umständlicher Bandtransport entfällt, Peripherie kann noch nach Ablauf des Programms zugeteilt werden
- Dialogbetrieb: vorher dauerte das Entdecken eines Syntaxfehlers oft Stunden

Geschichte der Betriebssysteme (6)

vierte Generation: (1985 - heute) PCs

- benutzerfreundlich, keine kryptische JCL (Job Control Language)
- im wesentlichen zwei Betriebssysteme: MS-Windows und verschiedene Ausprägungen von Unix (Solaris, HP-UX, Mac OS X, Linux, ...)
- Netzwerkbetriebssysteme sollen Peripherie auch anderen Rechnern nutzbar machen: Benutzer wissen von der Existenz anderer Rechner
- verteilte Betriebssysteme: erscheint den Benutzern wie ein Rechner, Transparenz: wo läuft das Programm, wo liegen die Daten

Linux: Wichtige Verzeichnisse

Dateisysteme aller Laufwerke sind Unterbäume des globalen Dateibaums. Funktionelle Gliederung:

`/`	Wurzelverzeichnis (root directory)
`/home`	(private) Verzeichnisse der Benutzer
`/dev`	Gerätedateien (Hardware-Komponenten)
`/etc`	Dateien zur Systemkonfiguration
`/bin`	essentielle Systemkommandos: `rm`, `mv`, `mount`, `netstat`
`/sbin`	entspricht `/bin`, dem Systemverwalter vorbehalten (**s**uper-user **bin**aries): `ifconfig`
`/usr/bin`	allgemein zugängliche Kommandos: `nslookup`
`/usr/sbin`	Systemverwalter vorbehaltene Kommandos: `lpd`, `ntpd`, `useradd`

Linux: Wichtige Verzeichnisse (2)

`/usr/include`	Header-Dateien für den C-Compiler
`/usr/doc`	verschiedene Dokumentationsdateien
`/usr/man`	Hilfe-Texte (manual pages)
`/usr`	Anwendungsprogramme und deren Daten
`/var`	Konfigurationsdateien
`/lib`	zum Systemstart notwendige Bibliotheken für dynamisch gelinkte Programme (shared libraries)
`/proc`	Pseudo-Dateisystem, Informationen über aktuellen Status
`/opt`	optionale Software (KDE, Netscape, ...)
`/mnt`	Dateisystem für wechselbare Medien

Linux: Gerätedateien

Zugriff auf Hardware-Komponenten über spezielle Einträge im Dateisystem. Einträge enthalten Gerätenummern, über die der Kernel die Gerätetreiber erreichen kann.

`/dev/fd0`	1. Floppy-Laufwerk
`/dev/fd1`	2. Floppy-Laufwerk
`/dev/hda`	1. AT-Bus Laufwerk (primary master)
`/dev/hdb`	2. AT-Bus Laufwerk (primary slave)
`/dev/hda1`	1. primäre Partition der 1. AT-Bus Festplatte
`/dev/hda5`	1. logische Partition der 1. AT-Bus Festplatte
`/dev/sda`	1. SCSI-Festplatte
`/dev/sdb1`	1. primäre Partition der 2. SCSI-Festplatte

Linux: Gerätedateien (2)

`/dev/cdrom`	Link auf das verwendete CD-ROM-Laufwerk
`/dev/mcd`	Mitsumi CD-ROM
`/dev/mouse`	Link auf die verwendete Maus-Schnittstelle
`/dev/psaux`	PS/2-Maus
`/dev/modem`	Link auf den Port, an dem das Modem angeschlossen ist
`/dev/ttyS0`	erste serielle Schnittstelle (COM1)
`/dev/ttyS1`	zweite serielle Schnittstelle (COM2)
`/dev/lp0`	parallele Schnittstelle (LPT1)

Linux: Virtuelle Konsolen

Im Textmodus stehen 6 virtuelle Konsolen zur Verfügung. Wechsel mittels Tastenkombination *Alt + F1* bis *Alt + F6*.

Die siebte Konsole ist für X11 reserviert: GUI, wurde 1984 am MIT entwickelt, die Version 11 hat sich als Standard etabliert, netzwerkfähig, Basis für Oberflächen wie KDE, Gnome.

Umschalten von X11 auf Textkonsole: *Ctrl + Alt + F1* bis *Ctrl + Alt + F6*

Zurückschalten auf X11: *Alt + F7*

Beenden des X-Servers: *Ctrl + Alt + BackSpace*

Linux: Hilfe-Texte

Über Befehle, Konfigurationsdateien und C-Bibliotheksfunktionen geben die **man pages** Auskunft:

`man -k` *name*	sucht nach *name* und listet die gefundenen Hilfe-Texte auf
`man` *name*	ruft den Hilfe-Text zu *name* auf

`apropos` sucht die Manualkurzbeschreibung in der Index-Datenbank (analog zu `man -k`).

Beispiel: `apropos login` liefert

```
logname (1)          - print user's login name
getlogin (3)         - get user name
sulogin (8)          - Single-user login
...
```

Linux: Dateisysteme

Linux unterstützt verschiedene Dateisysteme:

`affs`	Dateisystem des Amiga
`ext2`	Standard-Dateisystem unter Linux
`hpfs`	Standard-Dateisystem von IBM OS/2
`iso9660`	Standard-Dateisystem auf CD-ROMs
`nfs`	via Netzwerk zugängliches Dateisystem (network filesystem)
`ufs`	Dateisystem von BSD, SunOS und NeXTstep
`vfat`	Erweiterung des `fat`-Dateisystems (Microsoft Windows) hinsichtlich der Länge der Dateinamen

Linux: Dateisysteme (2)

`mount`: Datenträger in das Linux-Dateisystem einbinden.

Ablauf:

- `mount -t iso9660 /dev/cdrom /mnt` stellt den Inhalt der im CD-ROM-Laufwerk befindlichen CD im Verzeichnis `/mnt` bereit.
- Das Verzeichnis `/mnt` muss vorhanden sein.
- `umount /mnt` und `umount /dev/cdrom` entfernen das Dateisystem aus dem globalen Verzeichnisbaum. (Die CD-ROM kann nicht aus dem Laufwerk entfernt werden, solange das Verzeichnis gemountet ist.)

Linux: Zugriffsrechte auf Dateien

Nur der Systemadministrator (super user) `root` hat uneingeschränkte Zugriffsrechte auf alle Dateien, er darf als einziger *alle* Zugriffsrechte setzen oder löschen.

Diese Rechte gliedern sich in drei Teile: die Rechte des Besitzers, die Rechte für Gruppenmitglieder und die Rechte für alle anderen Systembenutzer.

Jede dieser drei Kategorien wird bei der Darstellung eines Verzeichniseintrags (`ls -l`) durch jeweils drei Zeichen angezeigt. Zusammen mit dem ersten Zeichen für den Dateityp ergeben sich 10 Flags für jede Datei.

Die möglichen Flags sind für die drei Kategorien gleich: `r` für lesbar (readable), `w` für schreibbar (writable) und `x` für ausführbar (executable).

Linux: Zugriffsrechte auf Dateien (2)

Das Zeichen - kennzeichnet ein nicht gesetztes Flag:

```
-rwxr-x---   1 rethmann cv    13332 Apr 19 12:01 kal
```

- Der Eigentümer (`rethmann`) darf die Datei lesen, ändern und ausführen (`rwx`),
- die Gruppenmitglieder (der Gruppe `cv`) dürfen die Datei lesen und ausführen (`r-x`),
- alle anderen Systembenutzer dürfen die Datei weder lesen, noch ändern, noch ausführen (`---`).

Zugriffsrechte können mittels des Komandos `chmod` geändert werden.

Linux: MS-DOS-Befehle

`mtools`: Speziell zum Bearbeiten von MS-DOS-Dateien.

- Alle `mtools`-Befehle heißen wie ihr DOS-Pendant, mit vorangestellten `m`.
- `mtools`-Befehle können nur verwendet werden, wenn die entsprechende Diskette *nicht* gemountet ist.

Wildcards müssen innerhalb von Anführungsstrichen stehen, sonst werden sie vom Kommando-Interpreter (Shell) interpretiert und nicht als Parameter übergeben:

```
mcopy "a:*.txt" /home/rethmann
```

kopiert alle Dateien mit Endung `.txt` vom ersten Diskettenlaufwerk in das Verzeichnis `/home/rethmann`.

Linux: Dienste

Linux bietet verschiedene Netzwerkdienste:

- Liste möglicher Dienste in `/etc/services`
- werden von Hintergrundprozessen, genannt Dämonen (daemon), verwaltet
- Programme werden bei Anforderung aktiv, führen die Kommunikation mit dem anfordernden System durch und die Dienstleistung aus.
- starten der Dienste entweder direkt oder mittels `inetd`
- Beispiele: ftp, telnet, ssh, daytime, http

Linux: Dienste (2)

inetd: It listens for connections on certain internet sockets. When a connection is found on one of its sockets, it decides what service the socket corresponds to, and invokes a program to service the request. After the program is finished, it continues to listen on the socket.

- Netzwerkdämon, verwaltet verschiedene Protokolle
- Konfiguration mittels `/etc/inetd.conf`
- einfache Dienste sind direkt implementiert: echo, time, daytime → Netzwerk-Handling nur einmal implementiert
- Dienstprogramme werden nur bei Bedarf geladen → Ressourcen schonen! Effizienz?

Linux: Dienste (3)

Bedeutung der Spalten in inetd.conf:

1. Service-Name (zugehöriger Port in `/etc/services`)
2. Socket-Typ: `stream` oder `dgram`
3. Protokoll: `tcp` oder `udp`
4. Flags: `wait` oder `nowait` (nach Anforderung wieder frei)
5. Privilegien für das Programm: `root`, `nobody`, ...
6. Pfad zum ausführenden Programm plus Argumente

Beispiele:

```
time    stream  tcp  nowait  root  internal
telnet  stream  tcp  nowait  root  /usr/sbin/tcpd  telnetd
```

Linux: Dienste (4)

tcpd: TCP-Wrapper

- Sicherheitsprogramm: prüfe, ob angeforderte Verbindung zulässig ist → entweder Verbindung beenden oder angegebenes Programm aufrufen
- feststellen der IP-Adresse des anfragenden Systems und vergleichen mit einem Regelsatz in `/etc/hosts.allow` und `/etc/hosts.deny`
- nähere Informationen in den man-pages über `tcpd(8)` und `hosts_acces(5)`

Linux: Dienste (5)

einfache Richtlinien:

- **Alles abschalten, was nicht unbedingt notwendig ist:** `echo`, `chargen`, `discard` usw. können durch gefälschte Pakete unnötig Last erzeugen
- **Alle r-Dienste abschalten:** `rlogin`, `rsh` usw. gelten als hochgradig unsicher

Mit einem Portscanner wie `nmap` können alle, auf einem Rechner aktivierten Dienste aufgelistet werden.

Linux: Dienste (6)

Starten von Diensten beim Systemstart:

- The scripts for controlling the system are placed in `/etc/init.d/`. These scripts are executed by `/sbin/init`.
- The configuration of `/sbin/init` is given by the file `/etc/inittab`.
- /sbin/init calls the run level master script `/etc/init.d/rc` to start or stop services provided by the other scripts under `/etc/init.d/`.
- To control the services of a run level, the corresponding scripts are linked into directories `/etc/init.d/rc<X>.d/`.

Linux: debian

Administration mittels `apt`:

1. `apt-setup` Sourcen auswählen, z.B. CD-Laufwerk und Internetadresse `http://www.debian.org`
2. `apt-update` download der Sourcen von CD oder aus dem Internet
3. `apt-get upgrade` normale Programme ersetzen
4. `apt-get dist-upgrade` Systemprogramme ersetzen
5. `apt-get install <xyz>` ein spezielles Paket einspielen
6. `apt-get clean` aufräumen

Mittels `dpkg-reconfigure xserver-xfree86` kann bspw. der X-Server neu konfiguriert werden, oder mittels `xf86config`.

Linux: debian (2)

Ermitteln der Hardware des Systems:

- `lspci` is a utility for displaying information about all PCI buses in the system and all devices connected to them.
- das `/proc`-Dateisystem enthält nützliche Einträge

weitere nützliche Kommandos:

- `ifconfig` zeigt die Konfiguration der Netzwerkkarte
- `route` legt die Gateways zum Erreichen anderer Sub-Netzwerke fest
- `ping` testet die Netzwerkverbindung auf unterster Ebene (ICMP echo request)

Prozesse

Ursprüngliche Ziele: die CPU voll ausnutzen und mehrere Benutzer gleichzeitig zulassen → Pseudo-Parallelität

Es ist auch konzeptionell sinnvoll, umfangreiche Aktivitäten in parallel ausführbare Teile zu zerlegen!

Prozess

- steht für die **Grundeinheit einer Aktivität** im System.
- ist der **Status** einer **Instanz** eines **Programms**:
  * Programm in Ausführung (laufend, bereit, blockiert)
  * zusammen mit allen Status-Informationen (Programmzähler, Registerinhalte, benutzte Ressourcen, Inhalte des Stacks/Heaps/Datensegments)

Prozesse (2)

Ein Programm ist statisch,

- eine Arbeitsvorschrift (Algorithmus)
- anwendbar auf beliebige Eingabedaten

Ein Prozess ist dynamisch,

- eine Aktivität (gesteuert von einer Arbeitsvorschrift)
- eine (sequentielle/parallele) Folge von Einzelaktivitäten
- besitzt einen Zustand, den Kontext des Programms
- ist festgelegt auf die aktuellen Eingabedaten
- zwei Prozesse können gleichzeitig dasselbe Programm ausführen

Prozesse (3)

Aufgaben des Betriebssystems:

- erzeugen neuer Prozesse
- beenden und entfernen alter Prozesse
- synchronisieren voneinander abhängiger Prozesse
- Kommunikation zwischen Prozessen

virtuelle Sicht: Prozess arbeitet mit virtueller CPU, über die er vollständige Kontrolle hat.

physische Sicht: Die CPU wird in kurzen Zeitabständen zwischen den Prozessen hin- und hergeschaltet.

Prozesse (4)

Prozess-Hierarchie:

- erzeugen eines Prozesses geschieht innerhalb eines anderen Prozesses (Eltern/Kind-Beziehung)
- in Unix: ein Vorgänger aller Prozesse (`/sbin/init`)

Ablauf der Prozess-Erzeugung in Unix:

- Prozess startet einen Kind-Prozess mittels `fork` → der aktuelle Prozess wird verdoppelt (identisch kopiert)
- beide Prozesse unterscheiden sich nur im Rückgabewert von `fork` (Kind: 0, Elter: PID des Kindes)
- üblicherweise überlädt sich der neue Prozess mit einem neuen Programm mittels `exec`

Prozesse (5)

```
#include <stdio.h>
#include <unistd.h>

/* Beschreibung: Das Programm ersetzt sich selbst durch
 * den ps-Befehl, indem es mittels exec den aktuellen
 * Prozess durch einen neuen Prozess ersetzt => Code, der
 * hinter dem exec-Aufruf steht, wird nicht ausgeführt!
 */
int main(int argc, char **argv) {
    printf("running ps with execlp ...\n");
    execlp("ps", "ps", "-a", 0);
    printf("done!\n");
    return 0;
}
```


Prozesse (6a)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

/* Beschreibung: Das Programm erzeugt eine Kopie von
 * sich selbst. Die Kopie wird mittels exec durch ein
 * anderes Programm ersetzt.
 */
int main(int argc, char **argv) {
    pid_t pid;

    printf("fork programm running ...\n");
    pid = fork();
```


Prozesse (6b)

```
    if (pid != 0) {          /* Eltern-Prozess */
        int i;
        printf("parent starting calc ...\n");
        for (i = 0; i < 10; i++) {
            printf("%d^2 = %d\n", i, i*i);
            sleep(1);
        }
    } else {                 /* Kind-Prozess */
        printf("child starting calc ...\n");
        execlp("_calc", "_calc", 0);
    }

    printf("done!\n");
    return 0;
}
```

Prozesse (7)

- **Programmende:** Es wird die Systemfunktion `_exit` aufgerufen (in C `exit(int ret)` oder `return int`)
- ret wird an den erzeugenden Prozess weitergegeben.
- Der Child-Prozess schickt dem Parent ein `SIGCHLD` und stribt erst, wenn der Parent-Prozess davon Kenntnis genommen hat.
- Bis dahin geht der Child-prozess in einen **Zombie**-Prozess über.
- `SIGCHLD` können auch ignoriert werden, dann geht aber auch der Rückgabewert verloren.
- Stirbt ein Parent-Prozess frühzeitig, wird der Child-Prozess unter Linux vom init-Prozess adoptiert.

Prozesse (8)

Beispiel: Als Benutzer an einem System anmelden.

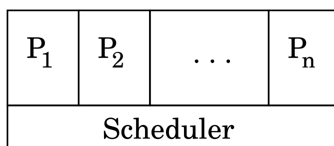
- Es wird angeschlossene Terminal bzw. virtuelle Konsole ein *getty*-Prozess gestartet
- *getty* erzeugt die Login-Meldung , nimmt den Benutzername und ersetzt sich durch das *login*-Kommando.
- Dieser Prozess nimmt das Passwort, verschlüsselt es und vergleicht es mit dem korrekten Password. Stimmen beide überein, so kann der Benutzer sich einloggen.
- *login* übernimmt noch einige Einstellungen und überlädt sich schließlich mit der für den Benutzer festgelegten Shell.

Prozesse (9)

Scheduler: Trifft die Entscheidung, welcher Prozess als nächstes die CPU erhält. Soll das Betriebsmittel CPU-Zeit möglichst fair vergeben.

traditionelle/externe Sicht:

- Scheduler ist unterste Schicht des Betriebssystems
- Benutzerprogramme und alle anderen BS-teile liegen darüber in Form von Prozessen



Minix: -Prozesse, die miteinander und mit den Benutzerprozessen kommunizieren -Kommunikation mittels Botschaftentransport

Prozesse: Linux/interne Sicht

- der Kernel ist (mehr oder weniger) monolithisch
- dispatcher: nimmt Umschaltung zwischen Prozessen vor
- scheduler: wählt nächsten auszuführenden Prozess aus
- Scheduler wird durch Timer-Interrupt aufgerufen
- Scheduler läuft im Interrupt-Kontext → hochpriorisierte Code-Sequenz, hat Zugriff Process-Control-Blocks (Datenstruktur: verkettete Liste von `task_struct`)
- User-Prozess: eigener Code + Dienste, die in seinem Auftrag vom Kernel abgearbeitet werden (system calls)
- Begriffe: Tasks und Threads sind Ausprägungen von Prozessen. jede Task hat ein eigenes Datensegment, mehrere Threads teilen sich ein Datensegment

Prozesse: Linux/interne Sicht (2)

- `time date` liefert:

```
Di Mär 23 22:33:49 CET 2004
0m0.081r  0m0.005u  0m0.003s
```

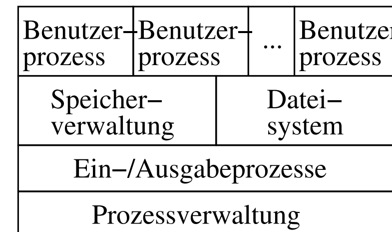
`u` gibt die Zeitdauer an, die die CPU mit dem eigenen Code, und `s` die Zeitdauer, die die CPU im Kernel (durch Erbringung eines angeforderten Dienstes) verbracht hat. Die Summe steht unter `r`.

- jeder Systemaufruf wird zuende abgearbeitet, ohne unterbrochen zu werden → ein Fehler in der Systemprogrammierung kann das ganze System blockieren

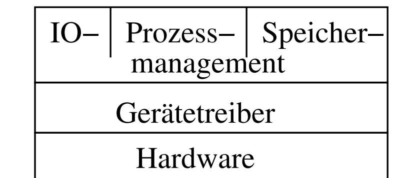
Prozesse: Minux vs. Linux

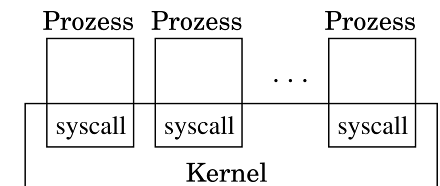
Tanenbaum/Minix:

Task
= Gerätetreiber
= Ein-/Ausgabeprozess



Linux:





Prozesse: Signale

Was ist das?

- Software-Entsprechung der Interrupt-Mechanismen
- Prozesse müssen auf eintreffende Signale reagieren
- Signale können zu jedem Zeitpunkt eintreffen
- einfache Art der Kommunikation zwischen Prozessen
- einzige übertragene Information: Nummer des Signals

Signale werden ausgelöst:

- in Fehlerfällen (Division durch 0)
- zum Mittteilen bestimmter Ereignisse (Timer-Ablauf)
- durch Benutzereingriff (`CTRL-C`, `CTRL-Z`, ...)
- durch den Systemaufruf `kill`

Prozesse: Signale (2)

ohne spezielle Maßnahmen ist kein Signal blockiert:

- alle Signale werden mit Standard-Reaktionen des Kernels beantwortet: `SIGSTOP` blockiert den Prozess, `SIGTERM` bricht den Prozess ab, ...
- mittels Systemaufruf `sigaction` kann eingestellt werden, wie auf ein Signal reagiert werden soll
- dazu werden Signal-Handler bei den Prozessen registriert

Prozesse: Signale (3)

einige POSIX-kompatible Funktionen/Systemaufrufe:

- `int sigemptyset(sigset_t *set)`
  leert die angegebene Signal-Menge
- `int sigfillset(sigset_t *set)`
  nimmt alle Signale in die Signal-Menge auf
- `int sigaddset(sigset_t *set, int signum)`
  nimmt das Signal `signum` in die Signal-Menge auf
- `int sigdelset(sigset_t *set, int signum)`
  löscht das Signal `signum` aus der Signal-Menge
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`
  ändert die Reaktion auf das Signal `signum`

Prozesse: Signale (4a)

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

/* --- KEIN ANSI-C --- Das Programm gibt in einer
 * Endlosschleife "Hello, world!" aus. Als Signal-
 * Routine für CRTL-C wird die Prozedur "handler"
 * registriert. Abbruch bspw. mit CTRL-\
 */

void handler(int sig) {
    printf("signal %d received\n", sig);
}
```


Prozesse: Signale (4b)

```
int main(int argc, char **argv) {
    struct sigaction action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGINT, &action, 0);

    while (1) {
        printf("Hello, world!\n");
        sleep(1);
    }
    return 0;
}
```


Prozesse: Pipes

Was ist das?

- Datenfluss von einem Prozess zu einem anderen
- bekannt aus der Verknüpfung zweier Befehle in der Shell: `ps -aux | grep httpd` → Ausgabe des einen Prozesses wird an die Eingabe des anderen weitergeleitet
- allgemein: dateiähnliche Verbindung zwischen Prozessen (Producer/Consumer), ein schreibender Kanal wird mit einem lesenden Kanal verbunden
- einfache Art, zwischen zwei Prozessen Informationen auszutauschen

Prozesse: Pipes (2)

POSIX-kompatible Funktionen

- `FILE *popen(const char *command, const char *type)`
  die Shell `/bin/sh` führt den String `command`, als Befehlszeile interpretiert, aus und koppelt seine Ein- (`type=r`) oder Ausgabe (`type=w`) an eine Pipe
- `int pclose(FILE *stream)`
  schließt eine mit `popen` geöffnete Pipe
- `int pipe(int filedes[2])`
  legt eine Pipe an und füllt das Array `filedes` mit zwei passenden File-Deskriptoren

Prozesse: Pipes (3a)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/* Mittels "popen" einen ls-Befehl absetzen und eine
 * Pipe aufbauen, um Ausgabe des ls-Befehls zu lesen.
 */
int main(int argc, char **argv) {
    FILE *pipe;
    char buf[BUFSIZ + 1];
    int cnt;
```


Prozesse: Pipes (3b)

```
    pipe = popen("ls -l", "r");
    if (pipe != 0) {
        printf("output was:\n");
        do {
            memset(buf, '\0', sizeof(buf));
            cnt = fread(buf, sizeof(char), BUFSIZ, pipe);
            printf("%s", buf);
        } while (cnt > 0);
        pclose(pipe);
    }
    return 0;
}
```


Prozesse: Pipes (4a)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

/* Eine Pipe aufbauen und mit zwei Prozessen verbinden.
 * Kind erzeugt Zahlen 0 bis 20, Elter liest die Daten,
 * gibt sie aus und wartet auf das Ende des Kindes.
 */
int main(int argc, char **argv) {
    int pid, files[2];

    pipe(files);
    pid = fork();
```

Prozesse: Pipes (4b)

```
if (pid != 0) {    /* Elter-Prozess: Consumer */
    char c;

    close(files[1]);
    while (read(files[0], &c, 1) > 0)
        write(STDOUT_FILENO, &c, 1);

    waitpid(pid, 0, 0);
}
```


Prozesse: Pipes (4c)

```
    else {           /* Kind-Prozess: Producer */
        char buf[8];
        int i;

        for (i = 0; i <= 20; i++) {
            sprintf(buf, "%d\n", i);
            write(files[1], buf, strlen(buf));
        }
    }
    return 0;
}
```


Prozesse: Named Pipes

Was ist das?

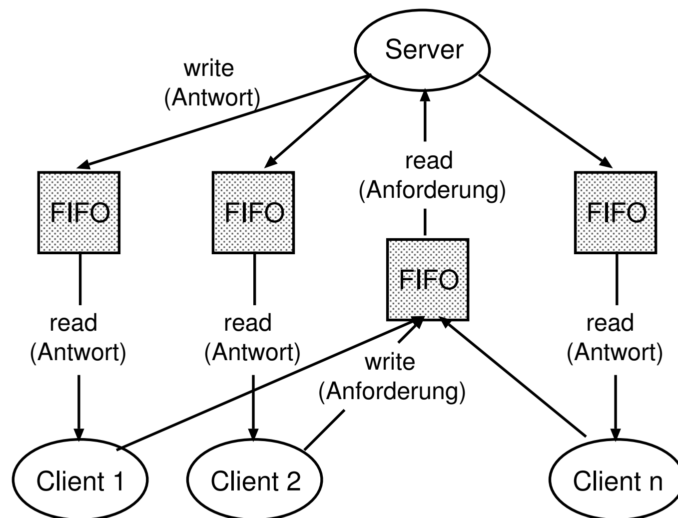
- Pipes können nur Daten zwischen verwandten Prozessen austauschen (Prozesse, die von einem gemeinsamen Vorgängerprozess gestartet wurden)
- named pipes bzw. FIFOs tauschen Daten zwischen beliebigen Prozessen
- spezielle Datei, die als Name im Dateisystem existiert, sich aber wie eine Pipe verhält
- können über die Befehlszeile oder innerhalb eines Programms erstellt werden: `mkfifo`
- ermöglicht Client/Server-Kommunikation

Prozesse: Named Pipes (2)

`int mkfifo(const char *pathname, mode_t mode)`
legt eine FIFO mit Pfad `pathname` im Dateisystem mit den Zugriffsrechten `mode & ~umask` an

- FIFO: öffnen mit `open`, schließen mit `close`
- es können mehrere Prozesse in eine FiFO schreiben und aus einer FIFO lesen
- Daten mehrerer schreibender Prozesse werden gemischt, Daten aus einem `write`-Aufruf stehen hintereinander
- FIFO zum Schreiben/Lesen öffnen, ohne dass ein lesender/schreibender Prozess existiert → `open` blockiert (blockieren beim Öffnen mit `O_NONBLOCK` vermeiden)
- wenn der letzte schreibende Prozess die FIFO schließt, erhalten die lesenden Prozesse ein EOF.

Prozesse: Named Pipes (3)



Prozesse: Named Pipes (4a)

```
#include <stdio.h> ...

#define REQFIFO_NAME "/tmp/primeserv_req"
#define ANSFIFO_NAME "/tmp/primeserv_ans"

/* Client-Prozess, der über eine Named Pipe/FIFO mit
 * dem Server kommuniziert. Zu testende Zahl wird als
 * Aufrufparameter übergeben. Requests: "pid:num"
 */
int main(int argc, char **argv) {
    FILE *cmdfifo, *ansfifo;
    pid_t pid;
    char ansfifo_name[256];
    int fd, num, answer;
```


Prozesse: Named Pipes (4b)

```
/* Request verschicken */
if ((fd = open(REQFIFO_NAME, O_WRONLY)) < 0) {
    fputs("server not running\n", stderr);
    exit(1);
}
cmdfifo = fdopen(fd, "w");
pid = getpid();
num = atoi(argv[1]);
fprintf(cmdfifo, "%d:%d\n", pid, num);
fflush(cmdfifo);
fclose(cmdfifo);
```


Prozesse: Named Pipes (4c)

```
    /* ggf. Antwort-FIFO anlegen und öffnen */
    sprintf(ansfifo_name, ANSFIFO_NAME ".%d", getpid());
    if (access(ansfifo_name, F_OK) < 0) {
        fprintf(stderr, "FIFO %s anlegen\n", ansfifo_name);
        if (mkfifo(ansfifo_name, 0666) < 0) {
            perror(ansfifo_name);
            exit(2);
        }
    }
    if ((fd = open(ansfifo_name, O_RDONLY)) < 0) {
        perror(ansfifo_name);
        exit(3);
    }
    ansfifo = fdopen(fd, "r");
```

Prozesse: Named Pipes (4d)

```
    /* Antwort auswerten */
    if (fscanf(ansfifo, "%d", &answer) != 1) {
        fprintf(stderr, "no answer from server\n");
    } else {
        if (answer == 0)
            printf("%d is not prime\n", num);
        else printf("%d is prime\n", num);
        fclose(ansfifo);
        unlink(ansfifo_name);
    }
    return 0;
}
```


Prozesse: Named Pipes (5a)

```
#define REQFIFO_NAME "/tmp/primeserv_req"
#define ANSFIFO_NAME "/tmp/primeserv_ans"
static FILE *cmdfifo;

/* Server-Prozess, der über eine named pipe/FIFO mit
 * den Clients kommuniziert. Server in Endlosschleife,
 * muss durch ein Signal abgebrochen werden.
 * Requests: "pid:num" Dienst: Primzahltest.
 */
void handler(int signo) {
   fclose(cmdfifo);
   unlink(REQFIFO_NAME);
   exit(0);
}
```


Prozesse: Named Pipes (5b)

```
short isPrime(unsigned long n) {
   int i, endval;

   endval = sqrt((double)n);
   for (i = 2; i <= endval; i++)
      if (n % i == 0)
         return 0;
   return 1;
}
```


Prozesse: Named Pipes (5c)

```
int main(int argc, char **argv) {
   int fd;

   signal(SIGINT, handler);
   if (access(REQFIFO_NAME, F_OK) < 0) {
      if (mkfifo(REQFIFO_NAME, 0666) < 0) {
         perror(REQFIFO_NAME);
         handler(SIGINT);
      }
   }
   if ((fd = open(REQFIFO_NAME, O_RDONLY)) < 0) {
      perror(REQFIFO_NAME);
      handler(SIGINT);
   }
   cmdfifo = fdopen(fd, "r");
```

Prozesse: Named Pipes (5d)

```
while (1) {
   char ansfifo_name[256], buf[256], *p = buf;
   int fd2, num;

   /* Anfrage auswerten */
   fgets(buf, 256, cmdfifo);
   p = strchr(buf, ':');
   if (p == 0) {
      fprintf(stderr, "illegal request: %s\n", buf);
      continue;
   }
```


Prozesse: Named Pipes (5e)

```
      /* Antwort verschicken */
      num = atoi(p+1);
      sprintf(ansfifo_name, ANSFIFO_NAME".%d", atoi(buf));
      if ((fd2 = open(ansfifo_name, O_WRONLY)) < 0) {
         fprintf(stderr, "can not open %s\n", ansfifo_name);
      } else {
         FILE *ansfifo = fdopen(fd2, "w");
         fprintf(ansfifo, "%d\n", isPrime(num));
         fclose(ansfifo);
      }
   }
   return 0;
}
```


Prozesse: Shared Memory

FIFOs sind zu langsam, um größere Datenmengen innerhalb eines Einprozessor-Systems auszutauschen → mache einzelne Speicher-Segmente mehreren Prozessen zugänglich:

- System verwaltet Segmente mittels IDs
- Prozess mit Segment verbinden mittels *attach*
- *attach* liefert Pointer, mit dem ganz normal auf den Speicher zugegriffen werden kann
- Segment vom Prozess abkoppeln mittels *detach*

Segmente werden bei `fork` an das Kind vererbt. Bei `exec` und `_exit` werden die Segmente abgekoppelt.

Prozesse: Shared Memory (2)

`int shmget(key_t key, int size, int flag)`
erfragt die ID eines bestehenden Segments oder legt ein neues an, `size` gibt die Größe des Segments in Bytes an.

`int shmctl(int id, int cmd, struct shmid_ds *buf)`
manipuliert ein Segment (`IPC_RMID` zum löschen).

`char *shmat(int id, char *addr, int flag)`
Ankoppeln des Segments `id` an den aktuellen Prozess. Rückgabe: Pointer zum Lesen/Schreiben des Segments. `addr` ist nur ein Adressvorschlag, der meißt ignoriert wird. `SHM_RDONLY` in `flag` schützt das Segment vor Schreibzugriffen.

`int shmdt(char *addr)`
Abkoppeln des Segments ab der Adresse `addr` vom aktuellen Prozess.

Prozesse: Shared Memory (3a)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/ipc.h>

/* Zwei Prozesse kommunizieren über einen gemeinsamen
 * Speicher: Writer schreibt Zufallszahlen, Reader
 * liest die Zahlen und gibt sie aus. Hier: Writer
 */
int main(int argc, char **argv) {
   double *data;
   int i, shmid;
   char buf[32];
```


Prozesse: Shared Memory (3b)

```
   shmid = shmget(IPC_PRIVATE, sizeof(double[256]), 0644);
   data = (double *)shmat(shmid, 0, 0);
   if (data == (double *)-1) {
      fprintf(stderr, "could not get shared memory\n");
      exit(1);
   }
   for (i = 0; i < 256; i++)
      data[i] = drand48();

   if (fork() == 0) {                /* Kind-Prozess */
      sprintf(buf, "%d", shmid);
      execlp("_shm_client", "_shm_client", buf, 0);
   }
   return 0;
}
```


Prozesse: Semaphore

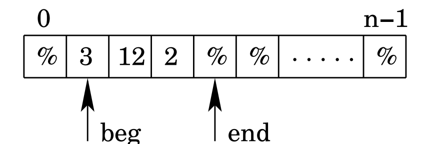
Es gibt vier grundsätzlich zu lösende Probleme bei Prozessen, die auf gemeinsam genutzte Ressourcen zugreifen:

- **Synchronisation:** eine bestimmte Operation in einem Prozess darf erst dann ausgeführt werden, wenn eine bestimmte Operation in einem anderen Prozess ausgeführt wurde (z.B. send/receive)
- **wechselseitiger Ausschluss:** nur ein Prozess darf zur selben Zeit Zugriff auf ein Objekt haben (z.B. Drucker)
- **Deadlocks:** Prozesse dürfen sich nicht gegenseitig endlos blockieren
- **Verhungern** (Starvation): ein Prozess darf nicht endlos auf seine benötigte Ressource warten müssen

Prozesse: Semaphore (2)

Listenverwaltung: Wettkampfbedingung

```
typedef struct {
    int cont[n];
    int beg, end;
} liste;
```

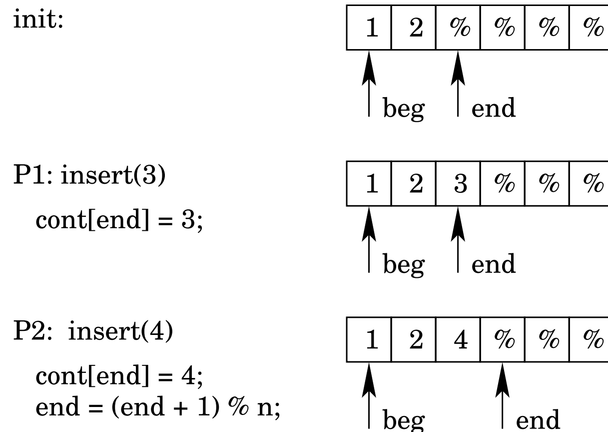



```
void insert(int val) {
    cont[end] = val;
    end = (end + 1) % n;
}
```

```
int next(void) {
    int r = cont[beg];
    beg = (beg + 1) % n;
    return r;
}
```

Prozesse: Semaphore (3)

Problem: Prozesse werden bei Ausführung eines Blocks zusammengehörender Operationen auf einem Objekt unterbrochen → inkonsistenter Zustand des Objekts



Prozesse: Semaphore (4)

Ein Semaphor ist eine ganzzahlige Variable, deren zusätzliche Funktionalität atomar/nicht teilbar ist. Es gibt genau zwei Operationen auf Semaphore:

- **down:** Ist der Wert des Semaphors größer als 0, dann wird der Wert um 1 erniedrigt. Ansonsten legt sich der aufrufende Prozess schlafen und wartet.
- **up:** Falls es einen Prozess gibt, der auf diesen Semaphor wartet, so wird er aufgeweckt. Sonst wird der Wert des Semaphors um 1 erhöht.

Anmerkung: Es wird keine Aussage darüber gemacht, welcher Prozess bei einem `up` aufgeweckt wird!

Prozesse: Semaphore (5)

Listenverwaltung bei mehreren Prozessen:

```
semaphor s = 1;

void insert(int val) {
   down(s);
   cont[end] = val;
   end = (end + 1) % n;
   up(s);
}

int next(void) {
   int r;

   down(s);
   r = cont[beg];
   beg = (beg + 1) % n;
   up(s);
   return r;
}
```

Frage: Funktioniert diese Lösung?

Prozesse: Semaphore (6)

UNIX: Funktionen und Strukturen in `sys/sem.h`, `sys/ipc.h`:

- `int semget(key_t key, int n, int flg)` erfragt die ID einer bestehenden Semaphor-Menge oder legt eine neue an. `n` gibt die Anzahl der Elemente an. Die Semaphor-Werte sind nicht initialisiert.
- `int semctl(int id, int num, int cmd, union semun arg)` führt Kontroll-Operation auf der Semaphor-Menge mit ID `id` aus. `num` selektiert einzelnen Semaphor, `cmd` wählt Kommando aus. `arg`: Argumente für das Kommando.
- `int semop(int id, struct sembuf *ops, unsigned int nops)` führt mehrere Semaphor-Operationen atomar aus. `ops`: Zeiger auf Array mit Operationsbeschreibungen. `nops`: Länge des Arrays. Prinzip: Alle oder keine Operation!

Prozesse: Semaphore (7)

Einige `cmd`-Parameter der Funktion `semctl`:

IPC_SET	Setzen von Besitzern und Zugriffsrechten
IPC_RMID	Löschen der Semaphor-Menge
GETVAL	Lesen eines Semaphor-Wertes
GETALL	Lesen aller Semaphor-Werte
SETVAL	Setzen eines Semaphor-Wertes

Prozesse: Semaphore (8)

Aufbau der verwendeten Struktur für Argumente von `cmd` aus `/usr/include/bits/sem.h`:

```
union semun {
  int val;                  // value for SETVAL
  struct semid_ds *buf;     // buffer for IPC_STAT & IPC_SET
  unsigned short *array;    // array for GETALL & SETALL
  struct seminfo *__buf;    // buffer for IPC_INFO
};
```


Prozesse: Semaphore (9)

`semop` führt up- und down-ähnliche Operationen aus:

```
struct sembuf {
   short sem_num;   // Nummer des Semaphors im Array
   short sem_op;    // Art der Operation
   short sem_flg;   // IPC_NOWAIT oder SEM_UNDO
};
```

`sem_op`: Wert, um den der Semaphor geändert werden soll.

- $>$ 0: entspricht **up**, Aufruf blockiert nie.
- = 0: Prozess blockiert, bis Semaphor-Wert ≥ 0 wird.
- $<$ 0: entspricht **down**. Aufruf blockiert, wenn Semaphor-Wert < 0 würde. Erst wenn Semaphor-Wert mindestens |`sem_op`| ist, wird die Änderung ausgeführt.

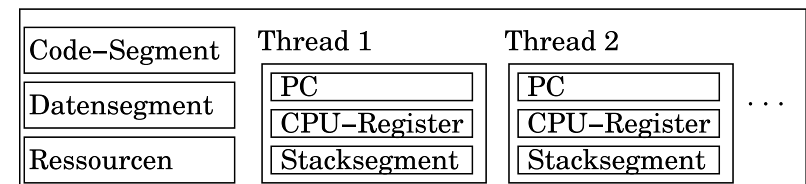
Threads

Ein Prozess ist klassischerweise die Grundeinheit für

- Ressourcen-Benutzung (Speicher, Dateien, ...)
- und CPU-Benutzung.

Thread: feinere Einheit, bezieht sich nur auf CPU-Nutzung

- sequentieller Strom von CPU-Aktivität innerhalb eines Prozesses
- Threads teilen sich die Ressourcen des Prozesses



Threads (3)

- Ein Prozess entsteht als **Klon** seines Erzeugers, ein Thread dagegen hat einen eigenen Körper, der in C in Form einer C-Funktion definiert wird.
- Jede Funktion, also auch die Thread-Funktion wird auf den Stack gelegt, d.h. die lokalen Daten sind privat, die globalen Daten stehen allen Threads zur Verfügung.
- Es gibt viele Thread-Varianten. Der Unix-Standard sind POSIX (Portable Operating System Interface) Threads oder PThreads.

Threads (4)

Anwendungen:

- Standard bei Entwurf komplexer Probleme wie BS
- notwendig zur Strukturierung vieler Probleme (z.B. GUI)
- Durchsatz kann erhöht werden: ein Thread wartet auf Beendigung des system calls, während ein anderer sinnvolle Dinge tut
- Einfache Möglichkeit der Programmbeschleunigung bei Mehrprozesor-Systemen
- Vorteil bei Server-Diensten: Gleichzeitig Anfragen von mehreren Prozessen/Rechnern beantworten, Code und die meisten Daten sind in allen Ausführungsströmen gleich

Threads (5)

- `int pthread_create(pthread_t *thread,`
  `pthread_attr_t *attr,`
  `void *(*start_routine)(void *), void *arg)`
  erzeugt eine neuen Thread und liefert dessen ID in `thread`. Der Thread führt zunächst die Funktion `start_routine` mit dem Argument `arg` aus.
- `void pthread_exit(void *retval)`
  wird aufgerufen, wenn ein Thread terminiert, entspricht `exit`-Aufruf bei Prozessen.
- `int pthread_join(pthread_t th, void **thread_return)`
  wartet auf den Thread `th` und dessen Rückgabewert `thread_return`, entspricht `wait`-Aufruf bei Prozessen.

Threads (6a)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

/* Programm startet Thread zur Berechnung des ggT,
 * wartet auf das Ergebnis und gibt es aus.
 */
int ggT(int *arg) {
    int p = arg[0];
    int q = arg[1];
    ...
    return q;
}
```

Threads (6b)

```
int main(int argc, char **argv) {
    int res, args[2];
    pthread_t aThread;
    void *threadResult;

    /* Programm-Argumente auswerten */
    if (argc != 3) {
        fprintf(stderr, "usage: %s val1 val2\n", argv[0]);
        exit(1);
    }
    args[0] = atoi(argv[1]);
    args[1] = atoi(argv[2]);
```


Threads (6c)

```
    res = pthread_create(&aThread, NULL,
              (void *(*)(void *))ggT, (void *)args);
    if (res != 0) {
        perror("Thread creation failed");
        exit(2);
    }
    printf("waiting for thread to finish...\n");
    res = pthread_join(aThread, &threadResult);
    if (res != 0) {
        perror("thread join failed");
        exit(3);
    }
    printf("thread returned %d\n", (int)threadResult);
    exit(0);
}
```


Threads (7)

Threads kommunizieren über gemeinsame Variablen und müssen synchronisiert werden. Linux unterstützt POSIX-kompatible Semaphore bisher **nur** bei Threads:

- `int sem_init(sem_t *sem, int flg, unsigned int val)` initialisiert den durch `sem` referenzierten Semaphor mit dem Wert `val`
- `int sem_wait(sem_t *sem)` entspricht **down**
- `int sem_post(sem_t *sem)` entspricht **up**
- `int sem_getvalue(sem_t *sem, int *sval)` auslesen des Wertes der durch `sem` referenzierten Semaphore

Threads (8a)

Datei `main.h`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#include "liste.h"
liste *lst;

/* Das Programm startet je einen Producer- und einen
 * Consumer-Thread. Der Producer schreibt zufällige
 * Daten in eine Liste, der Consumer entnimmt die Daten
 * aus der Liste und gibt sie aus.
 */
```

Threads (8b)

```
/* Producer: Items in die Liste einfügen */
int produce(void) {
    int i, item, end;

    printf("starting producer...\n");
    end = rand() % 8 + 4;
    for (i = 0; i < end; i++) {
        item = rand() % 1024;
        printf("put item %d\n", item);
        lst_insert(lst, item);
        sleep(rand() % 3);
    }
    return 0;
}
```


Threads (8c)

```
/* Consumer: Items aus der Liste entnehmen */
int consume(void) {
    int item;

    printf("starting consumer...\n");
    while (1) {
        printf("try to get item ...\n");
        item = lst_next(lst);
        printf("--> consumer got item %d\n", item);
        sleep(rand() % 3);
    }
    return 0;
}
```


Threads (8d)

```
int main(int argc, char **argv) {
    int res;
    void *threadResult;
    pthread_t consumer, producer;

    lst = (liste *)malloc(sizeof(liste));
    lst_init(lst);

    /* erzeuge Consumer-Thread */
    res = pthread_create(&consumer, NULL,
              (void *(*)(void *))consume, NULL);
    if (res != 0) {
        perror("Consumer creation failed");
        exit(1);
    }
```


Threads (8e)

```
    /* erzeuge Producer-Thread */
    res = pthread_create(&producer, NULL,
              (void *(*)(void *))produce, NULL);
    if (res != 0) {
        perror("Producer creation failed");
        exit(2);
    }

    /* warte auf Terminierung des Producers */
    printf("waiting for producer to finish...\n");
    res = pthread_join(producer, &threadResult);
    if (res != 0) {
        perror("producer join failed");
        exit(3);
    }
```

Threads (8f)

```
    /* Consumer beenden */
    printf("try to cancel consumer...\n");
    res = pthread_cancel(consumer);
    if (res != 0) {
        perror("Consumer cancelation failed");
        exit(4);
    }
    printf("waiting for consumer to finish...\n");
    res = pthread_join(consumer, &threadResult);
    if (res != 0) {
        perror("consumer join failed");
        exit(5);
    }
    return 0;
}
```


Threads (8g)

Datei `liste.h`

```
#include <semaphore.h>
#define BUF_SIZE 32

typedef struct {
    sem_t mutex;  /* wechselseitiger Ausschluss */
    sem_t free;   /* Anzahl freier Plätze in Liste */
    sem_t fill;   /* Anzahl belegter Plätze in Liste */
    int beg, end, cont[BUF_SIZE];
} liste;

void lst_insert(liste *l, int val);
int lst_next(liste *l);
void lst_init(liste *l);
```


Threads (8h)

Datei `liste.c`

```
#include "liste.h"

void lst_init(liste *l) {
    l->beg = 0;
    l->end = 0;
    sem_init(&l->mutex, 0, 1);
    sem_init(&l->free, 0, BUF_SIZE);
    sem_init(&l->fill, 0, 0);
}
```


Threads (8i)

```
void lst_insert(liste *l, int val) {
    sem_wait(&l->free);
    sem_wait(&l->mutex);
    l->cont[l->end] = val;
    l->end = (l->end + 1) % BUF_SIZE;
    sem_post(&l->mutex);
    sem_post(&l->fill);
}
```

Threads (8j)

```
int lst_next(liste *l) {
    int r;

    sem_wait(&l->fill);
    sem_wait(&l->mutex);
    r = l->cont[l->beg];
    l->beg = (l->beg + 1) % BUF_SIZE;
    sem_post(&l->mutex);
    sem_post(&l->free);
    return r;
}
```


Threads (9)

Weitere Synchronisationsmöglichkeiten:

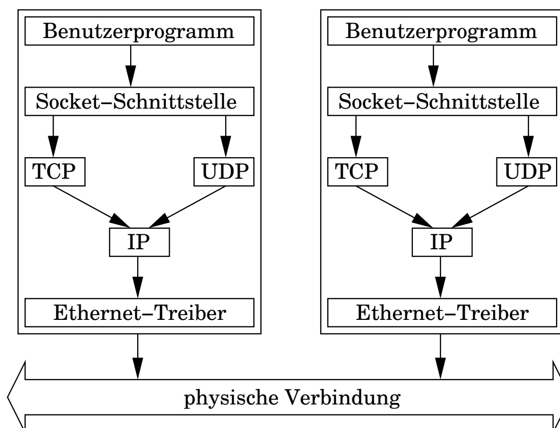
- Wechselseitiger Ausschluss:
  - `pthread_mutex_init()`
  - `pthread_mutex_lock()`
  - `pthread_mutex_unlock()`
- Bedingungsvariable:
  - `pthread_cond_init()`
  - `pthread_cond_wait()`
  - `pthread_cond_signal()`
  - `pthread_cond_broadcast()`

→ zur Übung

Sockets

Prozesse auf verschiedenen Rechnern kommunizieren in der Regel über Sockets (Steckdosen).

API für jedes Betriebssystem (Unix: Berkeley Socket Lib)



- ein Socket ist das Ende eines Kommunikationswegs
- Verbindungsdetails werden vom Kernel und Treibern übernommen
- Austausch von kleinen Paketen auf unterer Ebene

Anlegen von Sockets

- Die Socketschnittstelle ist analog zu einer Deskriptor-Tabelle für geöffnete Dateien angelegt.
- Ein **socket** ist eine interne Datenstruktur des BS zur Abarbeitung der Kommunikation eines Prozesses.
- Der **socket-Deskriptor** dieser Datenstruktur wird durch den Systemaufruf `socket` erzeugt, der einen **handle** für zukünftige Kommunikationsoperationen zurückgibt.
- Verschiedene Typen von Kommunikation unterscheiden sich insbesondere in der Zuverlässigkeit.

Anlegen von Sockets (2)

Include-Dateien: `sys/socket.h` (Socket-Definitionen), `netdb.h` (allgemeine Netzkommunikation) und `netinet/in.h` (Internet-Definitionen)

```
int socket(int domain, int type, int protocol)
```

- `domain`: Lebensbereich, der globale Einstellungen (Protokolle, Adressformate) vorgibt. Wichtige Konstanten: `AF_UNIX` und `AF_INET` (Linux-Versionen: `PF_UNIX`, `PF_INET`)
- `type`: Art der Verbindung. Beispiele: `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP) und `SOCK_RAW` (Roh-Daten, Umgehen von Netzwerk-Protokollen)
- `protocol`: sind in einer Domäne mehrere Protokolle erlaubt, kann eins ausgewählt werden. Normalerweise 0

Socket-Adressen

Sockets werden an eine bestimmte Adresse (Rechner und Port → Zugangspunkt für Netzwerkverbindung) gebunden.

Darstellung von Socket-Adressen: Struktur `sockaddr` oder spezielle Strukturen wie `sockaddr_in` für Internet-Sockets:

```
short            sin_family;  // Domain
unsigned short   sin_port;    // Port-Nummer
struct in_addr   sin_addr;    // Internet-Adresse
unsigned char    __pad[...];  // Füllen auf sockaddr-Größe
```

Server-Prozess: definiert eigenen Port und wartet auf Daten, die an diesen Port und Rechner geschickt werden.

Client-Prozess: muss die Port-Nummer des Servers kennen (`/etc/services`)

Socket-Adressen (2)

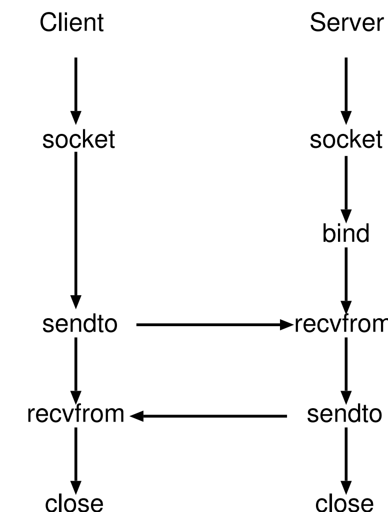
- Host-IP und Port müssen in Netzwerk-Ordnung umgeschrieben werden.
- Ports 1 bis 1023 gehören dem Superuser, 1024-49151 sind registrierte Ports, 49152 bis 65535 sind frei.

Bei Kommunikation innerhalb eines Rechner, wird die Unix-Domain verwendet:

```
short    sun_family;     // Domain AF_UNIX
char     sun_port[108];  // Path name
```

Weitere Domäne sind z.B. AF_APPLETALK oder AF_IPX

Verbindungslose Socket-Kommunikation



- Der Typ ist `SOCK_DGRAM`
- `bind` bindet eine Socket an eine Adresse
- Abschicken bzw. Empfangen von Daten: `sendto` / `recvfrom`
- close schließt den Socket
- Wird auf Grund der unzuverlässigkeit nicht in Client-Server Systemen verwendet.

Verbindungsorientierte Socket-Kommunikation

Bei Stream-Sockets wird mit Hilfe von `connect` und `accept` eine permanente Verbindung aufgebaut.

Datenaustausch geschieht über

- `int send(int s, const void *msg, size_t len, int flags)` schickt Daten der Länge `len` Bytes ab der Adresse `msg` über den Socket `s`. Liefert Anzahl geschriebener Bytes.
- `int recv(int s, void *buf, size_t len, int flags)` empfängt maximal `len` Bytes Daten über den Socket `s` und schreibt sie nach `buf`. Liefert Anzahl empfangener Bytes.

Datenaustausch über Sockets

Der `recv`-Aufruf blockiert, wenn keine Daten vorliegen. Aber mit `fcntl` (File Control Operations) kann abweichendes Verhalten definiert werden.

Liegen Daten von mehreren `send`-Aufrufen vor, können diese mit einem `recv`-Aufruf gelesen werden.

Wird der Flag auf `MSG_PEEK` gesetzt, werden die Daten gelesen ohne diese aus dem Socket zu entfernen.

Die Funktionen `read` und `write` entsprechen `recv` und `send`, aber ohne den Parameter `flags`.

Datenaustausch über Sockets (2)

```
int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout)
```

- wartet auf Status-Änderungen bei beliebig vielen File-Deskriptoren bzw. auf Anfragen bei Sockets (Bsp: `inetd`)
- Funktion blockiert, bis über einen Deskriptor `readfds`, `writefds` oder `exceptfds` Daten gelesen oder geschrieben werden können bzw. eine Datei ihren Status ändert.
- liefert die Anzahl der Deskriptoren, die das Ende von `select` ausgelöst haben.
- `timeout` begrenzt die Wartezeit (= 0 $\rightarrow$ unbegrenzt)

Datenaustausch über Sockets (3)

Zur Manipulation der Deskriptor-Mengen stehen Makros zur Verfügung: `FD_ZERO` leeren, `FD_SET` Deskriptor eintragen, `FD_CLR` Deskriptor austragen, `FD_ISSET` testet, ob der angegebene Deskriptor das Ende von `select` ausgelöst hat.

```
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Client-Sockets

Client-Socket: speziellen Host (Server) ansprechen, Kommandos schicken und Daten als Antwort empfangen. Auf dem entfernten Rechner muss bereits ein Socket existieren, mit dem der neue verbunden wird.

Aufbau einer Verbindung:

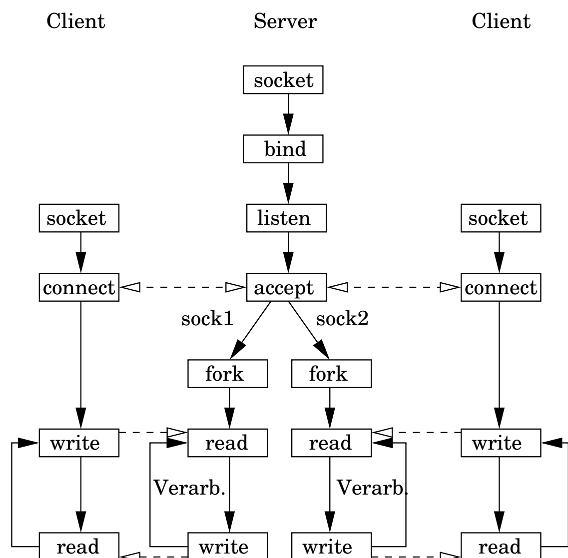
- Socket für gewünschtes Domain/Typ-Paar erzeugen
- numerische Adresse des Servers ermitteln (z.B. mittels `gethostbyname`)
- Adressstruktur wie `sockaddr_in` mit Informationen füllen
- Systemaufruf `connect` stellt Verbindung her

Server-Sockets

Listen-Socket ist Anlaufstelle für Client-Anfragen:

- Socket für gewünschtes Domain/Typ-Paar erzeugen
- in Adressstruktur lokale Adresse eintragen: entweder spezielle Netzwerkkarte eintragen oder angeben, dass auf allen Netzwerkkarten auf Anfragen gewartet werden soll
- der Systemaufruf `bind` bindet den Socket an die Adressstruktur (→ assigning a name to a socket)
- bei Stream-Sockets die Maximalgröße der internen Warteschlange für Anfragen mittels Systemaufruf `listen` angeben
- mittels Systemaufruf `accept` eine Client-Anfrage akzeptieren und accept-Socket erzeugen (accept blockiert, wenn keine Anfrage ansteht)

Server-Sockets (2)



Kommunikation mit Client über Accept-Socket

weitere Anfragen anderer Clients über Listen-Socket annehmen

Bearbeitung von Client-Aufträgen als Thread realisieren oder mittels `fork` abspalten

Sockets: benötigte Funktionen

- `struct hostent *gethostbyname(const char *name)` liefert eine Struktur `hostent` mit Informationen über den Host mit dem Namen `name`
- `int connect(int fd, struct sockaddr *addr, int len)` öffnet eine Verbindung vom Socket `fd` zu einem passenden Socket auf dem Server mit der Adresse `addr`.

  Da unterschiedliche Strukturen bei `addr` angegeben werden können (`sockaddr_in` bei Internet-Verbindungen), muss die Größe der Struktur in `len` übergeben werden.
- `int bind(int fd, struct sockaddr *addr, int len)` bindet den Socket `fd` an die lokale Adresse `addr`, Größe der Struktur in `len`

Sockets: benötigte Funktionen (2)

- `int listen(int s, int backlog)`
  definiert für den Socket `s` die Länge der Warteschlange für eingehende Verbindungen
- `int accept(int s, struct sockaddr *addr, int len)`
  akzeptiert Verbindung über Server-Socket `s`, erzeugt passenden Accept-Socket und gibt dessen Deskriptor zurück. `len` muss Länge der Adressstruktur enthalten. Füllt `addr` mit Informationen über anfragenden Client.
- `htonl`, `htons`, `ntohl`, `ntohs`
  convert value between host and network byte order
  * `unsigned short int htons(unsigned short int val)`
    converts `val` from host to network byte order
  * `unsigned short int ntohs(unsigned short int val)`
    converts `val` from network to host byte order

Sockets: network order

```
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
server_address.sin_port = 9734;.
```

Die Adresse ist die lokale Adresse (Rückkopplung), der Port wird auf 9734 gesetzt, jedoch:

```
my-pc> netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address         Foreign Address ...
tcp        0      0 localhost:mvel-lm     localhost:33786 ...
```

Aus /etc/services

```
mvel-lm         1574/udp .
```


Sockets: network order (2)

Es wird die Adresse `localhost:1574` verwendet und nicht `localhost:9734`.

Ursache: Auf Intel-Prozessoren ist $(9734)_{10} = (2606)_{16}$.

Werden die Bytes umgedreht, so ergibt sich X0626, was 1574 im Dezimalsystem entspricht.

TCP-IP verwendet **big endian**-Format (auch Most Significant Byte first), Intel **little endian** (auch Least Significant Byte first).

Die Funktion `htons` wandelt little endian in big endian um.

Sockets: network order (3)

- `int inet_aton(const char *cp, struct in_addr *inp)`
  converts the Internet host address `cp` from the standard numbers-and-dots notation into binary data and stores it in the structure that `inp` points to.
- `char *inet_ntoa(struct in_addr in)`
  converts the Internet host address given in network byte order into a string in standard numbers-and-dots notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.
- `in_addr_t inet_addr(const char *cp)`
  converts the Internet host address `cp` from numbers-and-dots notation into binary data in network byte order. If the input is invalid, `INADDR_NONE` is returned. This is an obsolete interface to `inet_aton`

Sockets: Beispiel

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* Alle Anfragen, die auf Port 8080 (HTTP) eintreffen,
 * protokollieren und als Antwort "I am alive" schicken.
 */
```


Sockets: Beispiel (2)

```
int main(int argc, char **argv) {
    int r, addrlen, aSocket, conn;
    int bufsize = 1024;
    int port = 8080;
    char *buf;
    struct sockaddr_in addr;

    /* create a server socket */
    aSocket = socket(PF_INET, SOCK_STREAM, 0);
    if (aSocket == -1) {
        perror("cannot open socket ");
        exit(1);
    }
```


Sockets: Beispiel (3)

```
    /* bind server port */
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);
    r = bind(aSocket, (struct sockaddr *)&addr,
             sizeof(addr));
    if (r == -1) {
        perror("cannot bind socket ");
        exit(2);
    }
```


Sockets: Beispiel (4)

```
    /* listen for incoming requests */
    listen(aSocket, 3);
    addrlen = sizeof(struct sockaddr_in);
    conn = accept(aSocket,
                  (struct sockaddr *)&addr, &addrlen);

    if (conn > 0)
        printf("client %s is connected...\n",
               inet_ntoa(addr.sin_addr));
```

Sockets: Beispiel (5)

```
    /* handle the request */
    buf = (char *)calloc(bufsize, sizeof(char));
    do {
        recv(conn, buf, bufsize, 0);
        printf("%s\n", buf);
    } while (strstr(buf, "\r\n\r\n") && strstr(buf, "\n\n"));

    send(conn, "\r\nI am alive", 12, 0);
    close(conn);

    free(buf);
    close(aSocket);

    return 0;
}
```