

# SysProg Übersicht

## Inhaltsverzeichnis

[Anpassen!]

- POSIX - Systemrufe und Fehlerbehandlung
  - Arbeit mit dem Dateisystem
  - Prozessverwaltung
  - Interprozesskommunikation
  - POSIX-Threads
  - Netzwerkprogrammierung
  - Linux - Gerätetreiber
- 

## man

- man pages eines UNIX-Systems sind i.d.R. in acht Abschnitte mit unterschiedlicher Ausrichtung unterteilt
- ist Stichwort in mehr als einem Abschnitt enthalten, so muss der gewünschte Abschnitt angegeben werden

Abschnitt	Inhalt
1	Nutzenkommandos wie ls, chown und less sowie Anwendungen
2	Systemaufrufe wie accept(), chown()
3	C Bibliotheks-Funktionen
4	Geräte- und Spezial-Dateien wie /dev/null, /dev/hdx, ...
5	Datei-Formate und Konventionen z.B. /etc/hosts, /etc/fstab u.a.
6	Spiele
7	Verschiedenes
8	Werkzeuge zur Systemverwaltung und Daemonen

---

## Allgemeine Workflow-Kommandos

---

Kommando	Funktion/Nutzen
ls -l	Größe Programm herausfinden
nm hello1	Ausgabe Symbole (Namen, Funktionen, Variablen)
ldd hello1	Anzeige Bibliotheken
man 3 printf	Informationen zur C-Funktion im UNIX-Manual ausgeben lassen
Strg + L	Refresh Screen
./scan	Programm scan in aktuellem Verzeichnis starten

## Compiler-Kommandos

Kommando	Funktion/Nutzen
gcc -o hello1 hello1.c	Programmübersetzung und Erzeugung eines ausführbaren Programmes hello1
gcc -g -o hello1 hello1.c	Programmübersetzung und Erzeugung ausführbaren Programmes mit Möglichkeit zur Analyse auf Quelltextniveau (für Debugging)
gcc -g -Wall -o hello1 hello1.c	-Wall schaltet alle Warnungen ein
gcc -g -Wall -o [name] [name.c]	Übersetzung mit sorgfältigeren Analyse (-Wall schaltet alle Warnungen 'ein')

### Weitere Kommandoschalter von gcc:

Schalter	Funktion/Nutzen
-o name	Name der ausführbaren Datei (sonst a.out)
-c name.c	übersetzt name.c nach name.o, kein Linken
-g	erzeugt debug-Informationen
-lbib	linkt Bibliothek <a href="#">libbib.so</a> oder libbib.a hinzu
-Ldir	sucht nach Bibliotheken auch in dir
-ldir	such nach Includes auch in dir
-Wall	schaltet alle wichtigen Warnungen ein

## Debuggen mit ***gdb***

- Debugger beherrschen, um in größeren Projekten effektiv und erfolgreich Fehler aufzuspüren! (Zeit für das Erlernen nehmen)
- Debugger gestatten: zeilenweise Abarbeitung von Programmen, Setzen von Haltepunkten, Inspizieren und Verändern von Variablen, ...
- gdb arbeitet kommandozeilenorientiert

Hinweis: um Debugger zu nutzen, muss Programm mit Informationen für Debugger übersetzt werden →  
`gcc -g -o [name] [name.c]`

Kommando	Funktion/Nutzen
<code>gdb hello1</code>	Debugger starten
<code>b 1 b main</code>	Haltepunkt auf Zeile 1 setzen (Ausführung endet an Breakpoint, nach run) breakpoint an Start main()
<code>r</code>	Programm hello1 starten (stoppt an ggf. gesetzten
<code>n</code>	eine Zeile ausführen/Ausführung nächste Anweisung
<code>q</code>	Debugger beenden
<code>strip hello1</code>	Debuginformationen und Symbole nachträglich entfernen
<code>gdb --tui [name]</code>	starte Debugger mit besseren Überblick
<code>s</code>	macht Programmschritt (in Funktion hinein)
<code>p [variable]</code>	Ausgabe variable

## Fehler und Warnungen

- aufgrund der Mehrdeutigkeit sollten Warnungen *immer* korrigiert werden

Fehler	Warnungen
Fehlermeldungen (errors)	Warnungen (warning)
syntaktische Fehler im Programm	deuten auf Unklarheiten in Interpretation des Quelltextes durch Compiler hin
Abbruch Übersetzung und Programm nicht erzeugt	i.d.R. Programm erzeugt, das jedoch fehlerhaft sein kann
Ergebnis: nichts zum Ausführen	Ergebnis: ausführbares Programm, ggf. fehlerhaft

## Makefiles und Make

- Ziel: Aufwand bei der Übersetzung eines Projektes minimieren
- Führt konfigurierbare Kommandos (Regeln) in Abhängigkeit von bestimmten Bedingungen (Abhängigkeiten) aus
- Besteht aus Abhängigkeiten und Regeln, die zur Lösung dieser Abhängigkeiten benutzt werden
  - Abhängigkeiten = Zusammenhänge zwischen Ziel- und Quell-Dateien (bspw. ziel: quelle1.o quelle2.o)
  - Regeln = Anweisungen, die auszuführen sind, um die Quellen in das Ziel umzuwandeln (bspw. `gcc -g -o ziel quelle1.o quell2.o`)
- Make-System führt insbesondere bei großen Projekten zu einer deutlichen Zeiteinsparung:

- Make-System erkennt anhand Zeitmarkierungen der beteiligten Quell-, Objekt- und ausführbaren Dateien, welche Teile des Gesamtprojektes verändert wurden und neu zu übersetzen sind
- Ausführung Makefile über Kommando *make*
  - führt Textdatei makefile im aktuellen Verzeichnis aus
  - andere Datei als Makefile nutzen:

```
make -f other_makefile
```

Beispiel: (siehe PR05)

```
# Make mehrere Programme, die im Rahmen des Praktikum 05 in SysProg erstellt wurden
all: prog-softlink copy create-softlink

# Make Softlink Programm
prog-softlink: softlink-main.o softlink.o
    gcc -o prog-softlink softlink-main.o softlink.o

softlink-main.o: softlink-main.c
    gcc -Wall -c softlink-main.c

softlink.o: softlink.c softlink.h
    gcc -Wall -c softlink.c

# Make sonstige Programme
copy: copy.c
    gcc -g -Wall -o copy copy.c

create-softlink: create-softlink.c
    gcc -g -Wall -o create-softlink create-softlink.c
```

## Fehlerbehandlung

Funktion	Wirkung
char *strerror(int errnum)	wandelt Fehlernummer in String
void perror(const char *msg)	gibt <i>msg</i> und Fehlerursache im Klartext aus

## Systemrufe zur Arbeit mit dem Dateisystem

### Filedeskriptoren

- repräsentieren geöffnete Datei
- Integer-Zahl 0 ... 1023

- vordefiniert und bei Programmstart geöffnet sind:  
0 (STDIN\_FILENO), 1 (STDOUT\_FILENO), 2 (STDERR\_FILENO)

## Öffnen, Erstellen, Lesen, Schreiben, Schließen

Funktion	Wirkung	Hinweis
<code>int open(const char *path, int flags, int mode);</code> <code>open(const char *path, int flags);</code>	Öffnen einer Datei, liefert kleinsten verfügbaren Filedeskriptor	nutze symbolische Flagnamen! mode beschreibt Zugriffsrechte, falls O_CREAT gesetzt
<code>int creat(const char *path, int mode);</code>	Erzeugen einer Datei	

Beachte Gleichwertigkeit:

```
int creat(const char *path, int mode);
int open(const char *path, O_CREAT | O_WRONLY | O_TRUNC, int mode);
```

<code>ssize_t read(int fd, void *buf, size_t nbytes)</code>	Lesen aus einer Datei	liefert Anzahl der tatsächlich gelesenen Bytes, 0 bei EOF, -1 bei Fehler
<code>ssize_t write(int fd, const void *buf, size_t nbytes);</code>	Schreiben in eine Datei	liefert Anzahl der geschriebenen Bytes
<code>off_t lseek(int fd, off_t offset, int whence);</code>	Positionieren des Dateizeigers	Werte für whence: SEEK_SET, SEEK_CUR, SEEK_END
<code>int close(int fd);</code>	Schließen der Datei	Filedeskriptor wird ungültig

## Symbolische Flagname für open()

- eines aus

O_RDONLY	nur lesen
O_WRONLY	nur schreiben
O_RDWR	lesen+schreiben

- bitweise ODER verknüpft mit

O_APPEND	schreiben hängt an, garantiert atomare Schreiboperationen
O_CREAT	anlegen, wenn Datei nicht vorhanden
O_EXCL	(nur in Verbindung mit O_CREAT!) Fehler, wenn Datei vorhanden
O_TRUNC	(vorhandene) Datei auf Länge 0 kürzen (in Verbindung mit O_CREAT, sodass existierende Datei überschrieben wird)
O_NONBLOCK	Rückkehr mit Fehler, wenn momentan keine Daten verfügbar sind (z.B. Netzwerk, serielle Schnittstelle)
O_SYNC	synchrons schreiben (kehrt erst zurück, wenn Daten auf Platte geschrieben sind)

## Abänderung Zugriffsrechte etc

Funktion	Wirkung	Hinweis
<code>int access(const char *pathname, int mode);</code>	Überprüfung der Zugriffsrechte	symbolische Zugriffsmodi nutzen (siehe man page)
<code>int chmod(const char *path, mode_t mode);</code> <code>int fchmod(int fd, mode_t mode);</code>	Änderung der Zugriffsrechte	
<code>int chown(const char *path, uid_t owner, gid_t group);</code>	Änderung des Eigentümers	auf vielen Systemen nur durch root möglich
<code>int umask(int mask);</code>	Setzen der Zugriffsrechtsmaske	gesetzte Bits werden bei den Systemrufen <code>open()</code> , <code>creat()</code> , <code>mkdir()</code> , <code>mkfifo()</code> aus Zugriffsrechten entfernt

## Abfrage Dateistatus

Funktion	Wirkung	Hinweis
<code>int stat(const char *filename, struct stat *buf);</code> <code>fstat(int fildes, struct stat *buf);</code>	Abfrage Dateistatus	Daten aus inode

vereinfachter Aufbau der *struct stat* unter Linux:

```
struct stat
{
    dev_t st_dev;           /* Device. */
    ino_t st_ino;           /* File serial number. */
    mode_t st_mode;         /* File mode. */
    nlink_t st_nlink;       /* Link count; */
    uid_t st_uid;           /* User ID of the file's owner. */
    gid_t st_gid;           /* Group ID of the file's group. */
    dev_t st_rdev;          /* Device number, if device. */
    off_t st_size;          /* Size of file, in bytes. */
    blksize_t st_blksize;   /* Optimal block size for I/O. */
    blkcnt_t st_blocks;     /* Number 512-byte blocks */
    time_t st_atime;         /* Time of last access. */
    time_t st_mtime;         /* Time of last modification. */
    time_t st_ctime;         /* Time of last status change. */
};
```

Folie 14 von 29

## Anlegen verschiedener Dateiarnten

## Sonstiges Dateioperationen

## Systemaufrufe zur Arbeit mit dem Dateisystem

Funktion	Bedeutung	Hinweise
<code>int open(const char *path, int flags, int mode);</code>	Öffnen einer Datei	liefert kleinsten verfügbaren Filedeskriptor
<code>int open(const char *path, int flags);</code>		
<code>int creat(const char *path, int mode);</code>	Erzeugen einer Datei	
<code>ssize_t read(int fd, void *buf, size_t nbytes);</code>	Lesen aus einer Datei	liefert Anzahl der tatsächlich gelesenen Bytes, d.h. 0 bei EOF, -1 bei Fehler
<code>ssize_t write(int fd, const void *buf, size_t nbytes);</code>	Schreiben in eine Datei	liefert Anzahl der geschriebenen Bytes
<code>off_t lseek(int fd, off_t offset, int whence);</code>	Positionieren des Dateizeigers	Werte für whence: SEEK_SET, SEEK_CUR, SEEK_END
<code>int close(int fd);</code>	Schließen einer Datei	Filedeskriptor wird ungültig
<code>int access(const char *pathname, int mode);</code>	Überprüfung der Zugriffsrechte	Beachte: symbolische Zugriffsmodi nutzen
<code>int chmod(const char *path, mode_t mode);</code>	Änderung der Zugriffsrechte	
<code>int fchmod(int fd, mode_t mode);</code>		
<code>int chown(const char *path, uid_t owner, gid_t group);</code>	Änderung des Eigentümers	auf vielen Systemen nur durch root möglich
<code>int umask(int mask);</code>	Setzen der Zugriffsrechtsmaske	gesetzte Bits werden bei den Systemrufen <code>open()</code> , <code>creat()</code> , <code>mkdir()</code> , <code>mkfifo()</code> aus Zugriffsrechten entfernt
<code>int stat(const char *filename, struct stat *buf);</code>	Abfrage des Dateistatus	Daten aus inode
<code>int fstat(int fildes, struct stat *buf);</code>		
<code>int link(const char *oldpath, const char *newpath);</code>	Erzeugen eines Hardlinks	
<code>int unlink(const char *pathname);</code>	Entfernen eines Hardlinks bzw. Löschen einer Datei	
<code>int rename(const char *oldpath, const char *newpath);</code>	Umbenennen einer Datei	
<code>int fcntl(int fd, int cmd);</code>	Ändern von Dateioptionen an geöffneten Dateien	
<code>int fcntl(int fd, int cmd, long arg);</code>		
<code>int dup(int oldfd);</code>	Duplizieren eines Filedeskriptors	liefert immer kleinstmöglichen

		Deskriptor zurück
<code>int mknod(const char *pathname, mode_t typ, dev_t dev);</code>	Anlegen einer Gerätedatei oder eines FIFO	
<code>int symlink(const char *oldpath, const char *newpath);</code>	Erzeugen eines symbolischen Links	Hinweis: Symbolischer Link == Softlink
<code>int readlink(const char *path, char *buf, size_t bufsiz);</code>	Lesen des Inhalts eines symbolischen Links	Dateiname, auf den Link verweist
<code>int lstat(const char *filename, struct stat *buf);</code>	Abfrage des Dateistatus des symbolischen Links selbst	

### Anwendung von dup()

```
fd=open("myfile.dat",O_RDONLY);
close(0);          /* schließt Standardeingabe          */
newfd = dup(fd);   /* newfd erhält Wert 0          */
close(fd);         /* alle Zugriffe auf stdin nach myfile.dat */
```

## Werte für flags

- eines aus

O_RDONLY	nur lesen
O_WRONLY	nur schreiben
O_RDWR	lesen+schreiben

- bitweise ODER verknüpfen mit

O_APPEND	schreiben hängt an, garantiert atomare Schreiboperationen
O_CREAT	anlegen, wenn Datei nicht vorhanden
O_EXCL	(nur in Verbindung mit O_CREAT) Fehler, wenn Datei vorhanden
O_TRUNC	(vorhandene) Datei auf Länge 0 kürzen
O_NONBLOCK	Rückkehr mit Fehler, wenn momentan keine Daten verfügbar sind (z.B. Netzwerk, serielle Schnittstelle o.Ä.)
O_SYNC	synchron schreiben (kehrt erst zurück, wenn Daten auf Platte geschrieben sind)

mode beschreibt Zugriffsrechte, falls O\_CREAT gesetzt

Beachte Gleichwertigkeit:

```
int creat(const char *path, int mode);
int open(const char *path, O_CREAT|O_WRONLY|O_TRUNC, int mode);
```

## Aufbau der struct stat unter Linux (vereinfacht)



```

struct stat
{
    dev_t st_dev;           /* Device. */
    ino_t st_ino;           /* File serial number. */
    mode_t st_mode;         /* File mode. */
    nlink_t st_nlink;       /* Link count; */
    uid_t st_uid;           /* User ID of the file's owner. */
    gid_t st_gid;           /* Group ID of the file's group. */
    dev_t st_rdev;          /* Device number, if device. */
    off_t st_size;          /* Size of file, in bytes. */
    blksize_t st_blksize;   /* Optimal block size for I/O. */
    blkcnt_t st_blocks;     /* Number 512-byte blocks */
    time_t st_atime;        /* Time of last access. */
    time_t st_mtime;        /* Time of last modification. */
    time_t st_ctime;        /* Time of last status change. */
};

```

The following mask values are defined for the file type:

```

S_IFMT      0170000  bit mask for the file type bit field

S_IFSOCK    0140000  socket
S_IFLNK     0120000  symbolic link
S_IFREG     0100000  regular file
S_IFBLK     0060000  block device
S_IFDIR     0040000  directory
S_IFCHR     0020000  character device
S_IFIFO     0010000  FIFO

```

Thus, to test for a regular file (for example), one could write:

```

stat(pathname, &sb);
if ((sb.st_mode & S_IFMT) == S_IFREG) {
    /* Handle regular file */
}

```

## Memory Mapped I/O

- Einblenden eines Teils einer Datei in den Adressraum eines Prozesses
- Typische Anwendungen: Shared Memory für verwandte Prozesse (erfordert Flag MAP\_SHARED)

Syntax	Funktion	Beschreibung	Return	Hinweise	Fehlersignale
--------	----------	--------------	--------	----------	---------------

Syntax	Funktion	Beschreibung	Return	Hinweise	Fehlersignale
void *mmap(void *start_addr, size_t length, int prot, int flags, int fd, off_t offset);	Blendet einen Teil der offenen Datei ein ab Stelle offset mit length Bytes in den Adressraum eines Prozesses ab Adresse start_addr	Einblendung in Adressraum eines Prozesses ab Adresse <b>*start_addr</b> <b>length</b> Anzahl Bytes <b>prot</b> E/A Schutzmodus <b>flags</b> <b>fd</b> File-Deskriptor <b>offset</b> ab Stelle Offset	gibt Anfangsadresse des gemappten Speicherbereiches zurück (MAP_FAILED = (void *)(-1)) bei Fehler	Datei muss in gewünschten Länge vorhanden sein (siehe ftruncate()) falls start_addr mit 0 vorgegeben → System legt geeignete Adresse fest prot muss mit Eröffnungsmodus für open() übereinstimmen (PROT_NONE, PROT_READ, PROT_WRITE, PROT_EXEC) um Datei in Speicher zu mappen, muss Datei lesbar sein	Fehler können Signale auslösen SIGSEGV Zugriff auf unerlaubten Speicherbereich SIGBUS Zugriff auf Speicherbereich, der nicht gemappt ist
int ftruncate(int fd, off_t length)	kürzt/erweitert offene Datei fd auf length Byte	bspw.: Datei in gewünschten Länge anlegen			
int munmap(void *start_addr, size_t length);	hebt Mapping auf			erneuter Zugriff liefert Segfault	