SysProg Übersicht

Inhaltsverzeichnis

[Anzupassen!]

---

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 libbib.so oder libbib.a hinzu
-Ldir	sucht nach Bibliotheken auch in dir
-Idir	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
gdb hello1	Debugger starten
b 1 b main	Haltepunkt auf Zeile 1 setzen (Ausführung endet an Breakpoint, nach run) breakpoint an Start main()
r	Programm hello1 starten (stoppt an ggf. gesetzten
n	eine Zeile ausführen/Ausführung nächste Anweisung
q	Debugger beenden
strip hello1	Debuginformationen und Symbole nachträglich entfernen
gdb --tui [name]	starte Debugger mit besseren Überblick
s	macht Programmschritt (in Funktion hinein)
p [variable]	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 beteiltigten 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 *msg* 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
int open(const char *path, int flags, int mode); int open(const char *path, int flags);	Öffnen einer Datei, liefert kleinsten verfügbaren Filedeskriptor	nutze symbolische Flagnamen! mode beschreibt Zugriffsrechte, falls O_CREAT gesetzt
int creat(const char *path, int mode);	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);
```

ssize_t read(int fd, void *buf, size_t nbytes)	Lesen aus einer Datei	liefert Anzahl der tatsächlich gelesenen Bytes, 0 bei EOF, -1 bei Fehler
ssize_t write(int fd, const void *buf, size_t nbytes);	Schreiben in eine Datei	liefert Anzahl der geschriebenen Bytes
off_t lseek(int fd, off_t offset, int whence);	Positionieren des Dateizeigers	Werte für whence: SEEK_SET, SEEK_CUR, SEEK_END
int close(int fd);	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 verfübar sind (z.B. Netzwerk, serielle Schnittstelle)
O_SYNC	synchrons schreiben (kehrt erst zurück, wenn Daten auf Platte geschrieben sind)

Abänderung Zugriffsrechte und co

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

Abfrage Dateistatus

Funktion	Wirkung	Hinweis
int stat(const char *filename, struct stat *buf); int fstat(int filedes, struct stat *buf);	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. */
  };
```

```
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 */
    }
```

Anlegen verschiedener Dateiarten

Funktion	Wirkung	Hinweis
int link(const char *oldpath, const char *newpath);	Erzeugen eines Hardlinks	Hardlink = mehrere Dateinamen verweisen auf gleichen i-Node/ gleiche Datei
int unlink(const char *pathname);	Entfernen eines Hardlinks bzw. Löschen einer Datei	
int mknod(const char *pathname, mode_t typ, dev_t, dev);	Anlegen einer Gerätedatei oder eines FIFO	Gerätedatei = Zugriff auf Gerätetreiber FIFO = FIFO-Speicher (spezielle Dateiform)
int symlink(const char *opath, const char *npath);	Erzeugen eines symbolischen Links	Symbolische-/Soft-Link = spezielle Dateiart, die Verweis auf andere Datei enthaelt
int readlink(const char *path, char *buf, size_t bufsiz);	Lesen des Inhalts eines symbolischen Links	= Dateiname, auf den Link verweist
int lstat(const char *filename, struct stat *buf);	Abfrage des Dateistatus des symbolischen Links selbst	

Sonstiges Dateioperationen

Funktion	Wirkung	Hinweis
int rename(const char *opath, const char *npath);	Umbenennen einer Datei	
int fcntl(int fd, int cmd); int fcntl(int fd, int cmd, long arg);	Ändern von Dateioptionen an geöffneten Dateien	
int dup(int oldfd);	Duplizieren eines Filedeskriptors	liefert immer kleinstmöglichen freien Deskriptor zurück

Anwendung von dup(): Eingabe-Umleitung aus einer Datei

```
fd = open("myfile.dat", O_RDONLY);
close(0); /* schliesst Standardeingabe */
newfd = dup(fd); /* liefert kleinstmoeglichen Deskriptor: newfd erhaelt Wert 0 */
close(fd); /* alle Zugriffe auf stdin nach myfile.dat */
```

Arbeit mit Verzeichnissen

- Verzeichnisse sind normale Dateien
- jedoch kein Schreiben möglich
    - außer durch Kernel, z.B. wenn Datei erzeugt/gelöscht/umbenannt wird

Funktion	Wirkung	Hinweis
int mkdir(const char *pathname, mode_t mode);	Erzeugen eines Verzeichnisses	
int rmdir(const char *pathname);	Löschen eines Verzeichnisses	
int chdir(const char *path);	Wechsel des aktuellen Verzeichnisses	Ausgangspunkt für relative Pfade
char *getcwd(char *buf, size_t size);	Abfrage des aktuellen Verzeichnisses	

Lesen des Verzeichnisinhaltes

Funktion	Wirkung	Hinweis
DIR *opendir(const char *name);	Öffnen eines Verzeichnisses	setzt Lesezeiger auf ersten Eintrag
struct dirent *readdir(DIR *dir);	Lesen des nächsten Verzeichniseintrages	Aufbau siehe unten, gibt NULL zurück am Verzeichnisende
void rewinddir(DIR *dir);	Rücksetzen des Verzeichnisses	
int closedir(DIR *dir);	Löschen des Verzeichnisses	

```
struct dirent {
    long            d_ino;       /* Inode Nummer */
    off_t           d_off;       /* Offset zum nächsten dirent */
    unsigned short  d_reclen;    /* Länge dieses Eintrags */
    char            d_name[NAME_MAX+1]; /* Dateiname */
};
```

Weiter ab 18 von 29

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
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 ***start_addr** **length** Anzahl Bytes **prot** E/A Schutzmodus **flags** **fd** File-Deskriptor **offset** 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	