U6 Verzeichnisse und Sortieren

- Linux-Benutzerumgebung
- Fehlerbehandlung
- POSIX-Verzeichnis-Systemschnittstelle
- Datei-Attribute in Inodes

U6-1 Linux-Benutzerumgebung

- Zugriff aus der Windows-Umgebung über
  - (1) SSH (nur Terminalfenster)
    - ◆ im Startmenü *Secure Shell Client* starten
    - ◆ verbinden mit einem beliebigen Linux-Rechner, z.B. *faui01*
    - ◆ Login mit den UNIX-Zugangsdaten
    - ◆ Editor: Notepad++
  - (2) Remote X (volle grafische Desktop-Umgebung)
    - ◆ im Startmenü im Menü *Linux-Terminal* einen beliebigen Rechner wählen
    - ◆ Login-Fenster erscheint, dort anmelden
    - ◆ danach ein Terminal-Fenster öffnen, z.B. *Konsole* unter KDE

1 Manual Pages

- aufgeteilt nach verschiedenen *Sections*
  - (1) Kommandos
  - (2) Systemaufrufe
  - (3) Bibliotheksfunktionen
  - (5) Dateiformate (spezielle Datenstrukturen, etc.)
  - (7) verschiedenes (z.B. Terminaltreiber, IP, ...)
- man-Pages werden normalerweise mit der Section zitiert: `printf(3)`

```
man [section] Begriff

z.B. man 3 printf
```

- Suche nach Sections: `man -f Begriff`
  Suche von man-Pages zu einem Stichwort: `man -k Stichwort`

2 Übersetzen und Ausführen

- spezielle Aufrufoptionen des Compilers

`gcc -pedantic`	liefert Warnungen in allen Fällen, die nicht 100% dem ANSI-C-Standard entsprechen
... `-Wall`	liefert in vielen evtl. zweifelhaften Situationen (die zwar korrekt sein könnten, aber häufig nicht sind) Warnungen

diese Optionen führen zwar oft zu nervenden Warnungen, helfen aber auch dabei, Fehler schnell zu erkennen:

```
gcc -pedantic -Wall -Werror -ansi -D_POSIX_SOURCE \
        -o printdir printdir.c
```

U6-2 Fehlerbehandlung

- Fehler können aus unterschiedlichsten Gründen im Programm auftreten
  - ➤ Systemressourcen erschöpft
    ➠ **malloc(3)** schlägt fehl
  - ➤ Fehlerhafte Benutzereingaben (z.B. nicht existierende Datei)
    ➠ **fopen(3)** schlägt fehl
  - ➤ Transiente Fehler (z.B. nicht erreichbarer Server)
    ➠ **connect(2)** schlägt fehl
- Gute Software **erkennt Fehler**, führt eine **angebrachte Behandlung** durch und gibt eine **aussagekräftige Fehlermeldung** aus
  - ◆ Kann das Programm trotz des Fehlers sinnvoll weiterlaufen?
  - ◆ Beispiel 1: Ermittlung des Hostnamens zu einer IP-Adresse für Logeintrag
    ➠ Fehlerbehandlung: IP-Adresse im Log eintragen, Programm läuft weiter
  - ◆ Beispiel 2: Öffnen einer zu kopierenden Datei schlägt fehl
    ➠ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden
    ➠ Oder den Kopiervorgang bei der nächsten Datei fortsetzen
    ➠ Entscheidung liegt beim Softwareentwickler

1 Fehler in Bibliotheksfunktionen

- Fehler treten häufig in Funktionen der C-Bibliothek auf
  - ➤ erkennbar i.d.R. am Rückgabewert (Manpage!)
- Die Fehlerursache wird meist über die globale Variable `errno` übermittelt
  - ➤ Fehlercode für jeden möglichen Fehler (siehe **errno(3)**)
  - ➤ Der Wert `errno=0` bedeutet Erfolg, alles andere ist ein Fehlercode
  - ➤ Bibliotheksfunktionen setzen `errno` im Fehlerfall
  - ➤ Bekanntmachung im Programm durch Einbinden von `errno.h`
- Fehlercodes können mit den Funktionen **perror(3)** und **strerror(3)** ausgegeben bzw. in lesbare Strings umgewandelt werden

```
char *mem = malloc(...); /* malloc gibt im Fehlerfall */
if(NULL == mem) {        /* NULL zurück */
    fprintf(stderr, "%s:%d: malloc failed with reason: %s\n",
        __FILE__, __LINE__-3, strerror(errno));
    perror("malloc"); /* Alternative zu strerror + fprintf */
    exit(EXIT_FAILURE); /* Programm mit Fehlercode beenden */
}
```


2 Erweiterte Fehlerbehandlung

- Signalisierung von Fehlern normalerweise durch Rückgabewert
- Nicht bei allen Funktionen möglich, z.B. **getchar(3)**

```
int c;
while ((c=getchar()) != EOF) { ... }

/* EOF oder Fehler? */
```

- Rückgabewert `EOF` sowohl im Fehlerfall als auch bei `End-of-File`

3 Fehlerbehandlung: Spezialfunktionen

- Erkennung im Fall von I/O-Streams mit **ferror(3)** und **feof(3)**

```
int c;
while ((c=getchar()) != EOF) { ... }

/* EOF oder Fehler? */
if(ferror(stdin)) {
    /* Fehler */
    ...
}
```

4 Fehlerbehandlung: direkte Verwendung von `errno`

- Nicht in allen Fällen existieren solche Spezialfunktionen
- Allgemeiner Ansatz durch Setzen und Prüfen von `errno`

```
#include <errno.h>
struct dirent *ent;
while (errno=0, (ent=readdir()) != NULL) {
    ... /* keine break-Statements in der Schleife */
}
/* EOF oder Fehler? */
if(errno != 0) {
    /* Fehler */
    ...
}
```

- `errno=0` *unmittelbar* vor Aufruf der problematischen Funktion
  ➥ `errno` wird nur im Fehlerfall gesetzt und bleibt sonst evtl. unverändert
- Abfrage der `errno` *unmittelbar* nach Rückgabe des pot. Fehlerwerts
  ➥ `errno` könnte sonst durch andere Funktion verändert werden

U6-3 POSIX-Verzeichnis-Systemschnittstelle

- Verzeichnisse öffnen: **opendir(3)**
- Verzeichnisse lesen: **readdir(3)**
- Verzeichnisse schließen: **closedir(3)**

1 opendir / closedir

- Funktions-Prototypen:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

- Argument von opendir
  - **`dirname`**: Verzeichnisname
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **`DIR`** oder **`NULL`**
- initialisiert einen internen Zeiger des directory-Funktionsmoduls auf den ersten Directory-Eintrag (für den ersten readdir-Aufruf)
- closedir schliesst ein geöffnetes Verzeichnis nach Bearbeitungsende

2 readdir

- liefert einen Directory-Eintrag (interner Zeiger) und setzt den Zeiger auf den folgenden Eintrag
- Funktions-Prototyp:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- Argumente
  - **`dirp`**: Zeiger auf **`DIR`**-Datenstruktur (von **opendir(3)**)
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **`struct dirent`** oder **`NULL`**, wenn **`EOF`** erreicht wurde oder im Fehlerfall
  - ➤ bei **`EOF`** bleibt **`errno`** unverändert (kritisch, kann vorher beliebigen Wert haben), im Fehlerfall wird **`errno`** entsprechend gesetzt
  - ➤ **`errno`** vorher auf 0 setzen, sonst kann **`EOF`** nicht sicher erkannt werden!

2 … readdir

- Problem: Der Speicher für die zurückgelieferte `struct dirent` wird von den dir-Bibliotheksfunktionen selbst angelegt und bei jedem Aufruf wieder verwendet!
  - ◆ werden Daten aus der dirent-Struktur länger benötigt, müssen sie vor dem nächsten readdir-Aufruf in Sicherheit gebracht (kopiert) werden
  - ◆ konzeptionell schlecht
    - ➤ aufrufende Funktion arbeitet mit Zeiger auf internen Speicher der readdir-Funktion
  - ◆ in nebenläufigen Programmen (mehrere Threads) nicht einsetzbar!
    - ➤ man weiß evtl. nicht, wann der nächste readdir-Aufruf stattfindet
- readdir ist ein klassisches Beispiel für schlecht konzipierte Schnittstellen in der C-Funktionsbibliothek

3 struct dirent

- Definition unter Linux (/usr/include/bits/dirent.h)

```
struct dirent {
    __ino_t d_ino;
    __off_t d_off;
    unsigned short int d_reclen; /* tatsächl. Länge der Struktur */
    unsigned char d_type;
    char d_name[256];
};
```

- POSIX: `d_name` ist ein Feld unbestimmter Länge, max. `NAME_MAX` Zeichen

U6-4 Datei-Attribute ermitteln: stat

- liefern Datei-Attribute aus dem Inode
- Funktions-Prototyp:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
```

- Argumente:
  - ◆ `path`: Dateiname
  - ◆ `buf`: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
printf("Inode-Nummer: %d\n", buf.st_ino);
```


1 stat: Ergebnisrückgabe im Vergleich zur readdir

- problematische Rückgabe auf funktions-internen Speicher wie bei `readdir` gibt es bei `stat` nicht
- Grund: `stat` ist ein Systemaufruf - Vorgehensweise wie bei `readdir` wäre gar nicht möglich
- der logische Adressraum des Anwendungsprogramms ist nur eine Teilmenge (oder sogar komplett disjunkt) von dem logischen Adressraum des Betriebssystems
  - ➤ Betriebssystemspeicher ist für Anwendung nicht sichtbar/zugreifbar
  - ➤ Funktionen des Kernels (wie stat) können keine Zeiger auf ihre internen Datenstrukturen an Anwendungen zurückgeben

2 stat / lstat: stat-Struktur

- `dev_t st_dev;` Gerätenummer (des Dateisystems) = Partitions-Id
- `ino_t st_ino;` Inodenummer (Tupel st_dev,st_ino eindeutig im System)
- **`mode_t st_mode;`** Dateimode, u.a. Zugriffs-Bits und Dateityp
- `nlink_t st_nlink;` Anzahl der (Hard-) Links auf den Inode
- `uid_t st_uid;` UID des Besitzers
- `gid_t st_gid;` GID der Dateigruppe
- `dev_t st_rdev;` DeviceID, nur für Character oder Blockdevices
- **`off_t st_size;`** Dateigröße in Bytes
- `time_t st_atime;` Zeit des letzten Zugriffs (in Sekunden seit 1.1.1970)
- `time_t st_mtime;` Zeit der letzten Veränderung (in Sekunden ...)
- `time_t st_ctime;` Zeit der letzten Änderung der Inode-Information (...)
- `unsigned long st_blksize;` Blockgröße des Dateisystems
- `unsigned long st_blocks;` Anzahl der von der Datei belegten Blöcke

3 stat / lstat: st_mode

- st_mode enthält Informationen über den Typ des Eintrags:
  - S_IFMT 0170000 bitmask for the file type bitfields
  - 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
- Zur einfacheren Auswertung werden Makros zur Verfügung gestellt:
  - S_ISREG(m) - is it a regular file?
  - S_ISDIR(m) - directory?
  - S_ISCHR(m) - character device?
  - S_ISLNK(m) - symbolic link? (Not in POSIX.1-1996.)