# Grundlagen der Verzeichnisbehandlung

Verzeichnisse werden ähnlich wie Dateien behandelt. Aus Sicht des Betriebssystems sind Verzeichnisse spezielle Dateien.

Die wichtigsten Befehle sind:

- `opendir()` zum Öffnen eines Verzeichnis,
- `closedir()` zum Schließen eines Verzeichnis und
- `readdir()` zum Lesen eines Verzeichnis.

## *Öffnen eines Verzeichnis*

**Syntax**

```
#include <dirent.h>
DIR *opendir(const char *dirname);
```

**Parameter**

- `dirname`: Name des zu öffnenden Verzeichnisses.

**Rückgabewert**

- Zeiger auf einen Verzeichnisstrom (vgl. `FILE`, Zeiger auf Dateistrom). Nach `opendir()` zeigt der Zeiger auf den ersten Eintrag.
- `NULL` wenn das Verzeichnis nicht geöffnet werden konnte (siehe `errno` für den Grund.)

## *Schließen eines Verzeichnis*

**Syntax**

```
int closedir(DIR *dirp);
```

Schließt ein geöffnetes Verzeichnis nach dessen Benutzung. Wie Dateien müssen auch Verzeichnisse wieder geschlossen werden!

**Parameter**

- `dirp`: Der Zeiger auf den Verzeichnisstrom.

**Rückgabewert**

- 0 bei Erfolg, sonst -1.

## *Lesen eines Verzeichniseintrags*

**Syntax**

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

**Parameter**

- `dirp`: Der Zeiger auf den Verzeichnisstrom. Der Inhalt des Zeigers wird um einen Eintrag erhöht.

**Rückgabewert**

- Zeiger auf ein Objekt vom Typ `struct dirent` oder
- `NULL` wenn kein weiterer Eintrag vorhanden ist (dann wird `errno` nicht verändert) oder
- bei einem Fehler (dann wird `errno` verändert).

Zur Fehleruntersuchung muss `errno` vor dem Aufruf auf 0 gesetzt werden, bei einem `NULL` -Zeiger kann dann auf Fehler kontrolliert werden (`errno != 0`).

Inhalt der Struktur `dirent`:

- `dirent` enthält die Informationen über den Verzeichniseintrag.
- Mindestens folgende Elemente sind vorhanden (abhängig vom Dateisystem können es mehr sein):

```
ino_t d_ino; /* Seriennummer der Datei (unter Unix die INode
Nummer.) */
char d_name[]; /* Der Name der Datei. */
```

## *Untersuchung der Dateieigenschaften*

Um die Eigenschaften einer Datei zu untersuchen, steht die `lstat()` Funktion zur Verfügung:

**Syntax**

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

**Parameter**

- `path`: Name der zu untersuchenden Datei (kann auch Verzeichnis, Link ... sein).
- `buf`: Zeiger auf ein Objekt vom Typ `struct stat`. Die Funktion `lstat()` beschreibt das Objekt mit den jeweiligen Dateiinformationen.

**Rückgabewert**

- 0 bei Erfolg, sonst -1.
- Die Struktur `stat` enthält alle Metainformationen über die Datei.

Der Inhalt von `struct stat` ist abhängig vom Dateisystem. Unter Unix gibt es u. a. folgende Strukturelemente:

```
mode_t st_mode /* Mode, Zugriffsrechte, .... */
nlink_t st_nlink; /* Zahl der Links auf diese Datei */
uid_t st_uid; /* User ID des Dateibesitzers */
gid_t st_gid; /* Group ID des Dateibesitzers */
off_t st_size; /* Dateigroesse in bytes */
time_t st_atime; /* Zeit des letzten Zugriffs */
time_t st_mtime; /* Zeit der letzten Modifikation */
time_t st_ctime; /* Zeit der letzten Statusaenderung */
```

```
long st_blksize; /* I/O Block Groesse */
```

Die Struktur `mode_t` definiert folgende Flags:

`S_IFIFO` : Die Datei ist eine *fifo* Spezialdatei.
`S_IFCHR` : Die Datei ist eine Zeichen Spezialdatei.
`S_IFDIR` : Die Datei ist ein **Verzeichnis**.
`S_IFBLK` : Die Datei ist eine *block* Spezialdatei.
`S_IFREG` : Die Datei ist eine **reguläre** Datei.

Folgende Testmakros sind für diese Flags definiert (Parameter `m` vom Typ `mode_t`):

`S_ISREG(m)` ist `m` eine **reguläre** Datei?
`S_ISDIR(m)` ist `m` ein **Verzeichnis**?
`S_ISCHR(m)` ist `m` ein *character* device (Treiber Datei)?
`S_ISBLK(m)` ist `m` ein *block* device (Treiber Datei)?
`S_ISFIFO(m)` ist `m` eine *fifo* Spezialdatei?
`S_ISLNK(m)` ist `m` ein **symbolischer Link**? (Nicht in POSIX.1-1996.)
`S_ISSOCK(m)` ist `m` eine *socket* Spezialdatei? (Nicht in POSIX.1-1996.)

Weitere Flags der `mode_t` Struktur:

`S_ISUID 04000` Setze die User Id des Besitzers bei Ausführung.
`S_ISGID 020#0` Setze die Group ID des Besitzers bei Ausführung.

Sind die `S_ISUID` und `S_ISGID` Flags gesetzt, kann ein beliebiger User die Datei mit den Rechten des Dateibesitzers ausführen. So kann z.B. auch ein User mit eingeschränkten Rechten Programme starten, die `root` Rechte benötigen (z. B. `mount` oder `su` Befehl).

`S_IRWXU 00700` Read, write, execute by owner.
`S_IRUSR 00400` Read by owner.
`S_IWUSR 00200` Write by owner.
`S_IXUSR 00100` Execute (search if a directory) by owner.
`S_IRWXG 00070` Read, write, execute by group.
`S_IRGRP 00040` Read by group.
`S_IWGRP 00020` Write by group.
`S_IXGRP 00010` Execute by group.
`S_IRWXO 00007` Read, write, execute (search) by others.
`S_IROTH 00004` Read by others.
`S_IWOTH 00002` Write by others
`S_IXOTH 00001` Execute by others.

# Referenzen


- Steve Grägert: POSIX-Programmierung mit UNIX, siehe Stud.IP
- W. Richard Stevens, Stephen A. Rago: Advanced Programming in the UNIX Environment. Second Edition, Addison-Wesley Professional, 2008. ISBN 0321525949
- M. Mitchell, J. Oldham, A. Samuel: Advanced Linux Programming. New Riders Publishing; 2001 (Verfügbar unter OSCA im Ordner Dokumente).