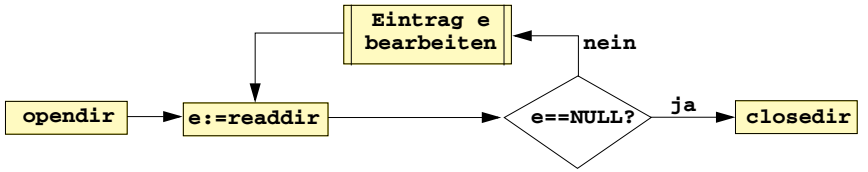
U6 Dateisystem

- Besprechung der 4. Aufgabe (clash)
- Aufbau eines UNIX-Dateisystems (Vorlesung *B VI-1 Seite 7ff*)
- Schnittstelle für Verzeichnisse
- Schnittstelle für Datei-Attribute
- Shell-Wildcards

U6-1 Verzeichnisse

- opendir(3), closedir(3)
- readdir(3), readdir_r(3)
- rewinddir(3)
- telldir(3), seekdir(3)

1 Iteratorkonzept zum Lesen von Verzeichnissen



2 opendir / closedir

- Funktions-Prototyp:

```
#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**
- Die **DIR**-Struktur ist ein Iterator und speichert die jeweils aktuelle Position
  - ◆ der Iterator steht nach Erzeugung auf dem ersten Verzeichniseintrag
- closedir gibt die mit belegten Ressourcen nach Ende der Bearbeitung frei

3 readdir

- liefert einen Verzeichniseintrag und setzt den **DIR**-Iterator auf den Folgeeintrag
- Funktions-Prototyp:

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

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

- Argumente
  - ◆ **dirp**: Zeiger auf **DIR**-Datenstruktur
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **struct dirent**, **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!

4 Fehlererkennung readdir...

- ...durch Setzen und Prüfen von `errno`

```
#include <errno.h>

while (errno=0, readdir(...) != NULL) {
    ... // keine break-Statements in der Schleife
}
// Ende 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

4 … readdir

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

5 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];
};
```

- Definition unter Solaris (/usr/include/sys/dirent.h)

```
typedef struct dirent {
    ino_t           d_ino;
    off_t           d_off;
    unsigned short  d_reclen; /* tatsächl. Länge der Struktur */
    char            d_name[1];
} dirent_t;
```

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

6 rewinddir

- setzt den **`DIR`**-Iterator auf den ersten Verzeichniseintrag zurück
  - ➤ nächster readdir-Aufruf liefert den ersten Verzeichniseintrag
- Funktions-Prototyp:

```
void rewinddir(DIR *dirp);
```

7 telldir / seekdir

- telldir ermittelt die aktuelle Position eines **`DIR`**-Iterator
- seekdir setzt einen **`DIR`**-Iterator auf einen zuvor abgefragten Wert
  - ◆ **`loc`** wurde zuvor mit telldir ermittelt
- Funktions-Prototypen:

```
long int telldir(DIR *dirp);
void seekdir(DIR *dirp, long int loc);
```

U6-2 Dateiattribute

- **stat(2)/lstat(2)** liefern Datei-Attribute aus dem Inode
- Funktions-Prototypen:

```
int stat(const char *path, struct stat *buf);

int lstat(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("Dateigroesse /etc/passwd: %ld\n", buf.st_size);
```


1 stat: Ergebnisrückgabe im Vergleich zu readdir

- problematische Rückgabe auf funktionsinternen Speicher wie bei **readdir** gibt es bei **stat** nicht
- Grund: **stat** ist ein Systemaufruf - Vorgehensweise wie bei **readdir** wäre gar nicht möglich
  - ➤ Vergleiche Vorlesung *B V-4* Seite 3
  - ➤ **readdir** ist komplett auf Ebene 3 implementiert (Teil der Standard-C-Bibliothek - Laufzeitbibliothek)
  - ➤ **stat** ist nur ein Systemaufruf(-stumpf), die Funktion selbst ist Teil des Betriebssystems (Ebene 2)
- der logische Adressraum auf Ebene 3 (Anwendungsprogramm) ist nur eine Teilmenge (oder sogar komplett disjunkt) von dem logischen Adressraum auf Ebene 2 (Betriebssystemkern)
  - ➤ Betriebssystemspeicher ist für Anwendung nicht sichtbar/zugreifbar
  - ➤ Funktionen der Ebene 2 können keine Zeiger auf ihre internen Datenstrukturen an Ebene 3 zurückgeben

1 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 (Vorl. 7-30)
- **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

1 stat-Zugriffsrechte

- in dem Strukturelement **st_mode** sind die Zugriffsrechte (12 Bit) und der Dateityp (4 Bit) kodiert.
- UNIX sieht folgende Zugriffsrechte vor (davor die Darstellung des jeweiligen Rechts bei der Ausgabe des ls-Kommandos)
  - **r** lesen (getrennt für *User*, *Group* und *Others* einstellbar)
  - **w** schreiben (analog)
  - **x** ausführen (bei regulären Dateien) bzw. Durchgriffsrecht (bei Dir.)
  - **s** setuid/setgid-Bit: bei einer ausführbaren Datei mit dem Laden der Datei in einen Prozess (exec) erhält der Prozess die Benutzer (bzw. Gruppen)-Rechte des Dateieigentümers
  - **s** setgid-Bit: bei einem Verzeichnis: neue Dateien im Verzeichnis erben die Gruppe des Verzeichnisses statt der des anlegenden Benutzers
  - **t** bei Verzeichnissen: es dürfen trotz Schreibrecht im Verzeichnis nur eigene Dateien gelöscht werden

U6-3 Shell-Wildcards

- Erlauben Beschreibung von Mustern für Pfadnamen
  `*`: beliebiger Teilstring (inkl. leerer String)
  `?`: genau ein beliebiges Zeichen
  `[a-d]`: ein Zeichen aus den Zeichen mit ASCII-Codes in ['a'; 'd']
  `[!a-d]`: ein Zeichen aus den Zeichen mit ASCII-Codes **nicht** in ['a'; 'd']
- Weitere und ausführliche Beschreibung siehe **glob(7)**
- Werden von der Shell expandiert, wenn im jeweiligen Verzeichnis passende Dateinamen existieren
  ☞ Quoting notwendig, wenn Muster als Argument übergeben wird
- Die Erweiterung betrifft immer nur einzelne Pfadkomponenten
- Dateien, die mit einem '.' beginnen, müssen explizit getroffen werden

1 Wildcard-Beispiel

```
mikey@lizzy[testdir] ls -a
attest.doc  t1.tar  t2.txt  test2.c  .test.c  test.c  tx.map
# Einfaches Teilstring-Wildcard
mikey@lizzy[testdir] ls test*
test2.c  test.c
# Mehrere Wildcards
mikey@lizzy[testdir] ls *test*
attest.doc  test2.c  test.c
# Einzelzeichen-Match
mikey@lizzy[testdir] ls test?.*
test2.c
# Bereiche
mikey@lizzy[testdir] ls t[1x].*
t1.tar  tx.map
# Invertierung eines Bereichs und Quoting
mikey@lizzy[testdir] find . -name 't[!12].*'
./tx.map
# Matching von Dateien, die mit einem .-Zeichen beginnen
mikey@lizzy[testdir] find . -name '.test*'
./.test.c
```


2 Evaluierung von Wildcard-Mustern in C-Programmen

- Funktion **fnmatch(3)**

```
#include <fnmatch.h>
int fnmatch(const char *pattern, const char *string, int flags);
```

- Prüft, ob das Wildcard-Muster `pattern` den String `string` einschließt
- Flags (0 oder bitweises Oder von ein oder mehreren der folgenden Werte)
  - `FNM_NOESCAPE`: Backslash als reguläres Zeichen interpretieren
  - `FNM_PATHNAME`: Ein Slash in `string` wird nur von einem Slash-Zeichen in `pattern` getroffen, nicht von einem Wildcard-Zeichen
  - `FNM_PERIOD`: Ein führender Punkt in einer Pfadkomponente muss von einem korrespondierenden Punkt in `pattern` getroffen werden
- Rückgabe
  - 0, wenn Muster den Teststring einschließt, sonst `FNM_NOMATCH`
  - andere Werte im Fehlerfall