

Übungen zu Systemprogrammierung 1 (SP1)

VL 6 – Dateisystem

Jens Schedel, Christoph Erhardt, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2012/13 – 17. Dezember bis 21. Dezember 2012

http://www4.cs.fau.de/Lehre/WS12/V_SP1

Agenda

- 6.1 Aufbau eines Dateisystems
- 6.2 Dateisystem-Schnittstelle
- 6.3 Wildcards
- 6.4 Gelerntes Anwenden



6.1 Aufbau eines Dateisystems

6.2 Dateisystem-Schnittstelle

6.3 Wildcards

6.4 Gelerntes Anwenden



Rechte auf Verzeichniseinträgen

- 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



- 6.1 Aufbau eines Dateisystems
- 6.2 Dateisystem-Schnittstelle
- 6.3 Wildcards
- 6.4 Gelerntes Anwenden



Dateiinformationen auslesen

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

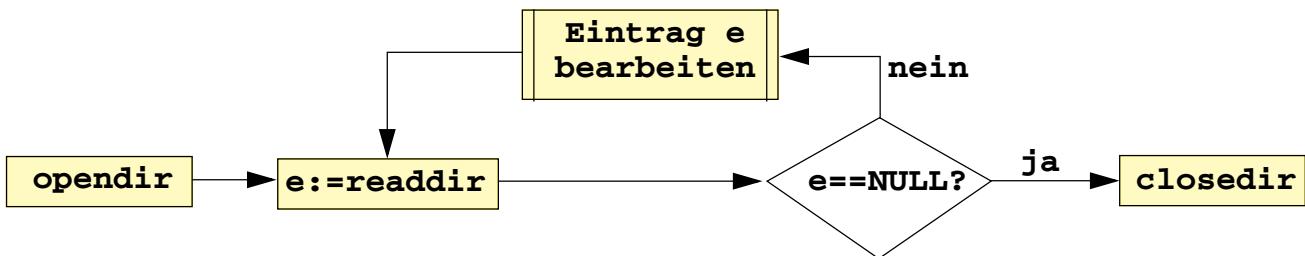
```
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

 - `path`: Dateiname
 - `buf`: Zeiger auf Puffer zum Speichern der Dateiinformationen
- Für uns relevante Strukturkomponenten der `struct stat`
 - `mode_t st_mode`: Dateimode, u.a. Zugriffs-Bits und Dateityp
 - Zur Bestimmung des Dateitypes gibt es u.A. folgende Makros: `S_ISREG`, `S_ISDIR`, `S_ISLNK`
 - `off_t st_size`: Dateigröße in Bytes



```
DIR *opendir(const char *dirname);  
struct dirent *readdir(DIR *dirp);  
int closedir(DIR *dirp);
```

- Die **DIR**-Struktur ist ein Iterator und speichert die jeweils aktuelle Position
- **readdir(3)** liefert einen Verzeichniseintrag und setzt den **DIR**-Iterator auf den Folgeeintrag
 - Rückgabewert **NULL** im Fehlerfall oder wenn EOF erreicht wurde
 - bei EOF bleibt **errno** unverändert, im Fehlerfall wird **errno** entsprechend gesetzt
- **closedir(3)** gibt die belegten Ressourcen nach Ende der Bearbeitung frei



Aufbau der Struktur struct dirent

```
struct dirent {  
    ino_t          d_ino;        /* inode number */  
    off_t          d_off;        /* offset to the next dirent */  
    unsigned short d_reclen;    /* length of this record */  
    unsigned char  d_type;      /* type of file; not supported  
                                by all file system types */  
    char           d_name[256];  /* filename */  
};
```

- **d_reclen**: Tatsächliche Länge der Struktur inklusive des Dateinamens
- **d_name**: Name des Verzeichniseintrages
- **d_type**: Eventuell Dateityp
 - **Nicht** verwenden, da nicht von allen Dateisystemen implementiert



- Der Speicher für die zurückgelieferte `struct dirent` wird von den Bibliotheksfunktionen selbst angelegt und beim nächsten `readdir`-Aufruf auf dem gleichen DIR-Iterator potentiell 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 nur bedingt einsetzbar
 - man weiß evtl. nicht, wann der nächste `readdir`-Aufruf stattfindet



Vergleich: readdir(3) und stat(2)

- Die problematische Rückgabe auf funktionsinternen Speicher wie bei `readdir(3)` gibt es bei `stat(2)` nicht
- Grund: `stat(2)` ist ein Systemaufruf - Vorgehensweise wie bei `readdir(3)` wäre gar nicht möglich
 - Vergleiche Vorlesung B V-4 Seite 3
 - `readdir(3)` ist komplett auf Ebene 3 implementiert (Teil der Standard-C-Bibliothek/Laufzeitbibliothek)
 - `stat(2)` 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



- 6.1 Aufbau eines Dateisystems
- 6.2 Dateisystem-Schnittstelle
- 6.3 Wildcards
- 6.4 Gelerntes Anwenden



Wildcards

- ...erlauben Beschreibung von Mustern für Pfadnamen
 - *: beliebiger Teilstring (inkl. leerer String)
 - ? : genau ein beliebiges Zeichen
 - [a-d] : ein Zeichen aus den Zeichen a - d
 - [!a-d] : ein Zeichen nicht aus den Zeichen a - d
- Die Erweiterung betrifft immer nur einzelne Pfadkomponenten
- Dateien, die mit einem '.' beginnen, müssen explizit getroffen werden
- 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



- ...mit der Funktion `fnmatch(3)`

```
int fnmatch(const char *pattern, const char *string, int flags);
```

- Prüft, ob der String `string` zum Wildcard-Muster `pattern` passt.
- Flags (0 oder bitweises Oder von ein oder mehreren der Werte)
 - `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
 - Weitere Flags siehe Man-Page



Agenda

- 6.1 Aufbau eines Dateisystems
- 6.2 Dateisystem-Schnittstelle
- 6.3 Wildcards
- 6.4 Gelerntes Anwenden



„Aufgabenstellung“

- Fun with Wildcards
- Ausgabe aller Dateinamen von symbolischen Verknüpfungen im aktuellen Verzeichnis

