

Übungen zu Systemprogrammierung 1

Ü6 – Dateisystem

Wintersemester 2022/23

Luis Gerhorst, Jonas Rabenstein, Dustin Nguyen, Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



7.1 Aufbau eines Dateisystems

7.2 Dateisystem-Schnittstelle

7.3 Wildcards

7.4 Gelerntes anwenden



7.1 Aufbau eines Dateisystems

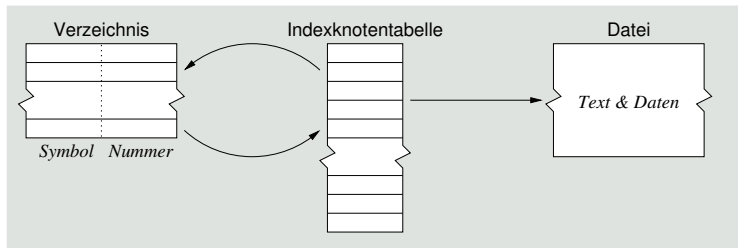
7.2 Dateisystem-Schnittstelle

7.3 Wildcards

7.4 Gelerntes anwenden

Datenstrukturen im Namensraum⁹

Dateisystem (*file system*)



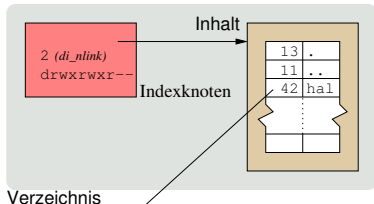
- die **Indexknotentabelle** (*inode table*) ist ein statisches Feld (*array*) von Indexknoten und die zentrale Datenstruktur
 - ein Indexknoten ist **Deskriptor** insb. eines Verzeichnisses oder einer Datei
- das **Verzeichnis** (*directory*) ist eine **Abbildungstabelle**, es übersetzt symbolisch repräsentierte Namen in Indexknotennummern
 - eine von der Namensverwaltung des Betriebssystems definierte Datei
- die **Datei** (*file*) ist eine abgeschlossene Einheit zusammenhängender Daten beliebiger Repräsentation, Struktur und Bedeutung

⁹Als Einheit auf demselben Medium (z.B. Ablagespeicher) abgelegt.

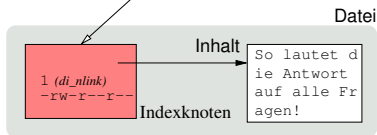


Verzeichniseintrag II

- ein Namenverzeichnis ist eine **spezielle Datei** der Namensverwaltung



- das selbst einen Namen hat, der einen Indexknoten bezeichnet
- über eine Verknüpfung erreichbar ist aus einem anderen Verzeichnis
- Namen getrennt von eventuellen Dateiinhalten speichert



*Verknüpfungen anlegen/löschen zu können, ist eine **Berechtigung**, die sich nur auf das Verzeichnis der betreffenden Verknüpfungen bezieht!*

- Selbstreferenz („dot“, 13) und Elterverzeichnis („dot dot“, 11) geben wenigstens zwei Verweise auf ein Verzeichnis
 - auch wenn das Verzeichnis selbst sonst keine weiteren Namen enthält





- 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 Verzeichnissen)
 - 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



```
> ls -oaiR .
.:
total 16
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 .
886 drwxr-xr-x 3 dust 4096 Jul 12 17:09 ..
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 narf
612 -rw-r--r-- 1 dust    7 Jul 12 22:32 .private

./narf:
total 16
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 .
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 ..
611 -rw-r--r-- 2 dust    3 Jul 12 22:41 bar
611 -rw-r--r-- 2 dust    3 Jul 12 22:41 foo
613 lrwxrwxrwx 1 dust   11 Jul 12 22:40 public -> ../.private
```



```
> ls -oaiR .
.:
total 16
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 .
886 drwxr-xr-x 3 dust 4096 Jul 12 17:09 ..
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 narf
612 -rw-r--r-- 1 dust    7 Jul 12 22:32 .private

./narf:
total 16
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 .
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 ..
611 -rw-r--r-- 2 dust    3 Jul 12 22:41 bar
611 -rw-r--r-- 2 dust    3 Jul 12 22:41 foo
613 lrwxrwxrwx 1 dust   11 Jul 12 22:40 public -> ../.private
```

Indexknoten

| | |
|----------|------|
| st_ino | 609 |
| st_nlink | 3 |
| st_size | 4096 |

Inhalt

```
609 .
886 ..
610 narf
612 .private
```




```
> ls -oaiR .
.:
total 16
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 .
886 drwxr-xr-x 3 dust 4096 Jul 12 17:09 ..
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 narf
612 -rw-r--r-- 1 dust 7 Jul 12 22:32 .private
./narf:
total 16
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 .
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 ..
611 -rw-r--r-- 2 dust 3 Jul 12 22:41 bar
611 -rw-r--r-- 2 dust 3 Jul 12 22:41 foo
613 lrwxrwxrwx 1 dust 11 Jul 12 22:40 public -> ../.private
```

Indexknoten

| | |
|----------|-----|
| st_ino | 612 |
| st_nlink | 1 |
| st_size | 7 |

Inhalt

<7 Byte Inhalt>




```
> ls -oaiR .
.:
total 16
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 .
886 drwxr-xr-x 3 dust 4096 Jul 12 17:09 ..
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 narf
612 -rw-r--r-- 1 dust 7 Jul 12 22:32 .private

./narf:
total 16
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 .
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 ..
611 -rw-r--r-- 2 dust 3 Jul 12 22:41 bar
611 -rw-r--r-- 2 dust 3 Jul 12 22:41 foo
613 lrwxrwxrwx 1 dust 11 Jul 12 22:40 public -> ../.private
```

Indexknoten

| | |
|----------|------|
| st_ino | 610 |
| st_nlink | 2 |
| st_size | 4096 |

Inhalt



```
610 .
609 ..
611 bar
611 foo
613 public
```



```
> ls -oaiR .
.:
total 16
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 .
886 drwxr-xr-x 3 dust 4096 Jul 12 17:09 ..
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 narf
612 -rw-r--r-- 1 dust    7 Jul 12 22:32 .private

./narf:
total 16
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 .
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 ..
611 -rw-r--r-- 2 dust    3 Jul 12 22:41 bar
611 -rw-r--r-- 2 dust    3 Jul 12 22:41 foo
613 lrwxrwxrwx 1 dust   11 Jul 12 22:40 public -> ../.private
```

Indexknoten

| | |
|----------|-----|
| st_ino | 611 |
| st_nlink | 2 |
| st_size | 3 |

Inhalt

<3 Byte Inhalt>



```
> ls -oaiR .
.:
total 16
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 .
886 drwxr-xr-x 3 dust 4096 Jul 12 17:09 ..
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 narf
612 -rw-r--r-- 1 dust    7 Jul 12 22:32 .private

./narf:
total 16
610 drwxr-xr-x 2 dust 4096 Jul 12 22:41 .
609 drwxr-xr-x 3 dust 4096 Jul 12 22:40 ..
611 -rw-r--r-- 2 dust    3 Jul 12 22:41 bar
611 -rw-r--r-- 2 dust    3 Jul 12 22:41 foo
613 lrwxrwxrwx 1 dust   11 Jul 12 22:40 public -> ../.private
```

Indexknoten

| | |
|----------|-----|
| st_ino | 613 |
| st_nlink | 1 |
| st_size | 11 |

Inhalt

../.private



7.1 Aufbau eines Dateisystems

7.2 Dateisystem-Schnittstelle

7.3 Wildcards

7.4 Gelerntes anwenden



- `stat(2)/lstat(2)` liefern Datei-Attribute aus dem Inode
- Unterschiedliches Verhalten bei Symlinks:
 - `stat(2)` folgt Symlinks (rekursiv) und liefert Informationen übers Ziel
 - `lstat(2)` liefert Informationen über den Symlink selber

■ Funktions-Prototypen

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

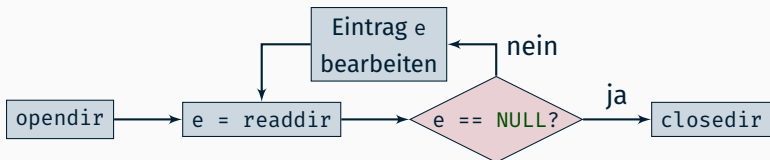
```
int lstat(const char *path, struct stat *buf);
```

- `path`: Dateiname
- `buf`: Zeiger auf Puffer zum Speichern der Dateiinformatoren
- 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);
```

- DIR-Struktur ist ein Iterator und speichert 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





- Verzeichniseintrag

```
struct dirent {  
    ino_t d_ino;    /* inode number */  
    char  d_name[]; /* filename */  
};
```

- Struct hat in Linux weitere Felder, bspw. d_type
Sind nicht in POSIX definiert, dürfen in SP **nicht** verwendet werden



- 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



- 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.2 Seite 19ff.
 - `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



7.1 Aufbau eines Dateisystems

7.2 Dateisystem-Schnittstelle

7.3 Wildcards

7.4 Gelerntes anwenden



- ... erlauben Beschreibung von Mustern für Pfadnamen
 - * beliebiger Teilstring (inklusive 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
 - 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



| | test* | *test* | test?.* | t[1x].* | t[!12].* | .test* |
|------------|--------------|---------------|----------------|----------------|-----------------|---------------|
| .test.c | | | | | | X |
| attest.doc | | X | | | | |
| t1.tar | | | | X | | |
| t2.txt | | | | | | |
| test.c | X | X | | | | |
| test2.c | X | X | X | | | |
| tx.map | | | | X | X | |



- ... 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)
 - `FNМ_PATHNAME`: Ein Slash in `string` wird nur von einem Slash-Zeichen in `pattern` getroffen, nicht von einem Wildcard-Zeichen
 - `FNМ_PERIOD`: Ein führender Punkt in einer Pfadkomponente muss von einem korrespondierenden Punkt in `pattern` getroffen werden
 - Weitere Flags siehe Man-Page



7.1 Aufbau eines Dateisystems

7.2 Dateisystem-Schnittstelle

7.3 Wildcards

7.4 Gelerntes anwenden



„Aufgabenstellung“

- Ausgabe aller Dateinamen von symbolischen Verknüpfungen im aktuellen Verzeichnis