Übungen zu Betriebssysteme

Ü8 – Dateisystem

Sommersemester 2023

Henriette Hofmeier, Manuel Vögele, Benedict Herzog, Timo Hönig

Bochum Operating Systems and System Software Group (BOSS)

Faculty of Computer Science

RUHR UNIVERSITÄT BOCHUM

RUB

Agenda

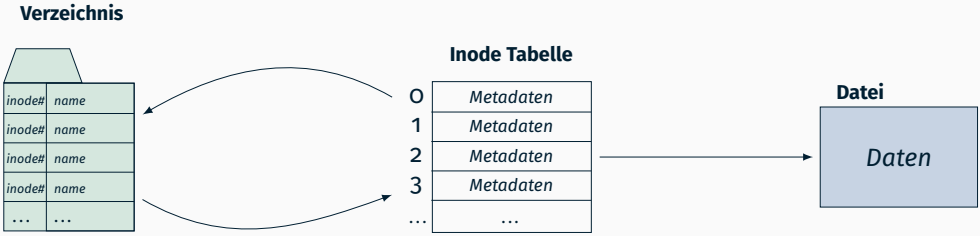
8.1 Aufbau eines Dateisystems

8.2 Dateisystem-Schnittstelle

8.3 Gelerntes anwenden

Agenda

Dateien und Verzeichnisse (UNIX)



- **Inode Tabelle:** zentrale Datenstruktur
  - Inode enthält Metadaten zu einer Datei/einem Verzeichnis
  - z.B.: Zugriffsrechte, Besitzer, Größe, Datenblöcke, …
- **Verzeichnis:** Abbildungstabelle
  → Inhalt: bildet Namen auf Inode-Nummer ab
- **Datei:** zusammenhängende Daten
  → Inhalt: beliebiger Inhalt

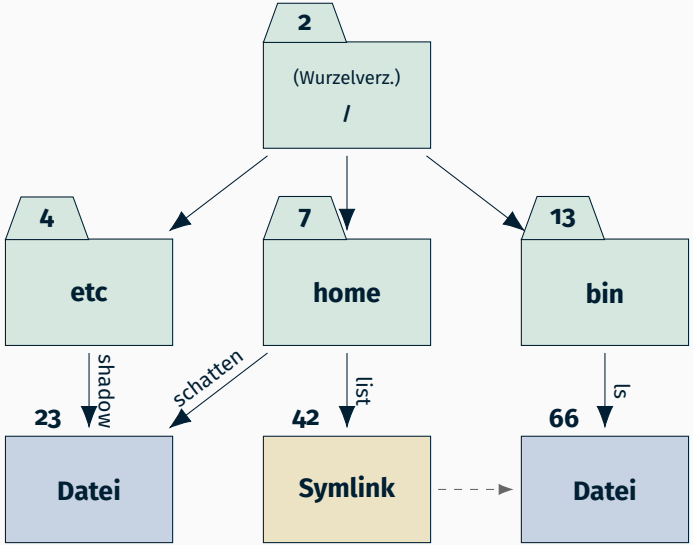
Hard- und Symbolic Links (UNIX)

- **Hardlink**
  - mehrere Verzeichniseinträge auf selbe Datei/Verzeichnis (Inode)
    - → Zugriff mit unterschiedlichen Namen/Pfaden aber identischen Inhalt
    - → Name unterschiedlich, restlichen Metadaten (Zugriffsrechte, Eigentümer, etc.) identisch
  - `rm <file>` löscht Eintrag aus Verzeichnistabelle und dekrementiert Hardlink-Zähler
    - → Datei/Verzeichnis wird gelöscht, wenn alle Verweise gelöscht wurden
  - nur innerhalb des selben Dateisystems möglich
    - → Warum?

Hard- und Symbolic Links (UNIX)

- **Hardlink**
  - mehrere Verzeichniseinträge auf selbe Datei/Verzeichnis (Inode)
    - → Zugriff mit unterschiedlichen Namen/Pfaden aber identischen Inhalt
    - → Name unterschiedlich, restlichen Metadaten (Zugriffsrechte, Eigentümer, etc.) identisch
  - `rm <file>` löscht Eintrag aus Verzeichnistabelle und dekrementiert Hardlink-Zähler
    - → Datei/Verzeichnis wird gelöscht, wenn alle Verweise gelöscht wurden
  - nur innerhalb des selben Dateisystems möglich
    - → Warum?
- **Symbolic Link (Symlink)**
  - hat eine eigene Inode (mit eigenen Metadaten)
  - Inhalt/Daten der Inode ist Pfad zum eigentlichen Ziel
  - `rm <symlink>` löscht Symlink (Ziel bleibt unberührt)
  - Ziel kann in anderem Dateisystem sein
    - Warum?
    - Was passiert wenn das Ziel gelöscht wird?

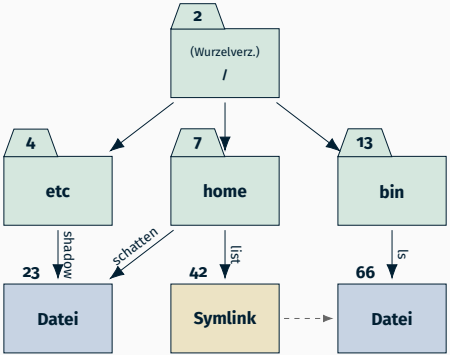
Beispiel Dateisystem (UNIX)



Beispiel Dateisystem (UNIX)

Wurzelverzeichnis

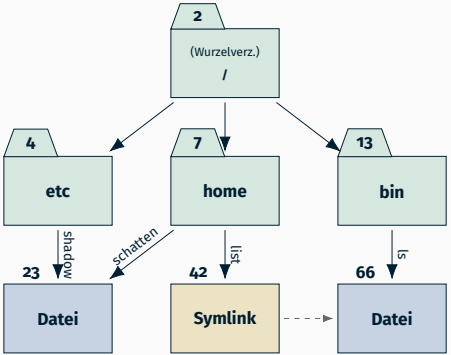
2	
?	.
?	..
?	etc
?	home
?	bin



Beispiel Dateisystem (UNIX)

Wurzelverzeichnis

2	
2	.
2	..
?	etc
?	home
?	bin



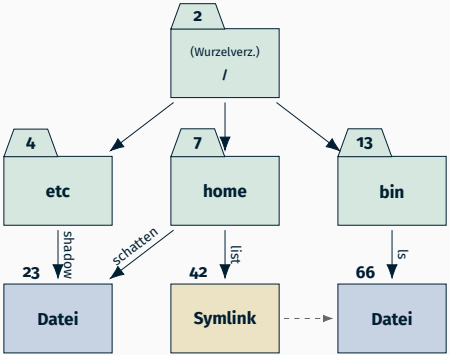
Beispiel Dateisystem (UNIX)

Wurzelverzeichnis

2	
2	.
2	..
4	etc
7	home
13	bin

etc

4	
?	.
?	..
?	shadow



Beispiel Dateisystem (UNIX)

Wurzelverzeichnis

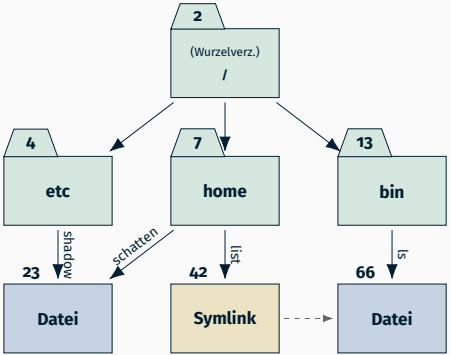
2	
2	.
2	..
4	etc
7	home
13	bin

etc

4	
4	.
2	..
23	shadow

home

7	
?	.
?	..
?	schatten
?	list



Beispiel Dateisystem (UNIX)

Wurzelverzeichnis

2	
2	.
2	..
4	etc
7	home
13	bin

etc

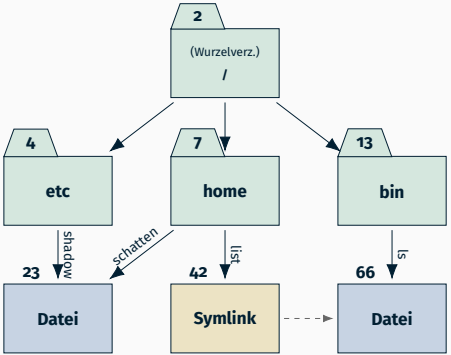
4	
4	.
2	..
23	shadow

home

7	
7	.
2	..
23	schatten
42	list

bin

13	
13	.
2	..
?	ls



Beispiel Dateisystem (UNIX)

Wurzelverzeichnis

2

2	.
2	..
4	etc
7	home
13	bin

etc

4

4	.
2	..
23	shadow

home

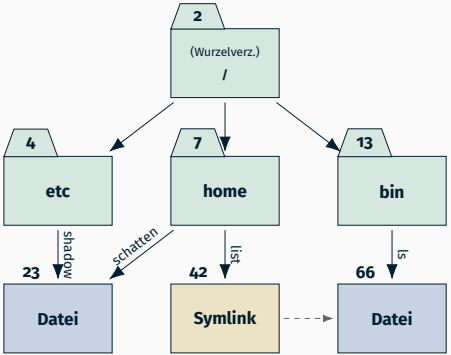
7

7	.
2	..
23	schatten
42	list

bin

13

13	.
2	..
66	ls



Beispiel Dateisystem (UNIX)

Wurzelverzeichnis

2

2	.
2	..
4	etc
7	home
13	bin

etc

4

4	.
2	..
23	shadow

home

7

7	.
2	..
23	schatten
42	list

bin

13

13	.
2	..
66	ls

23

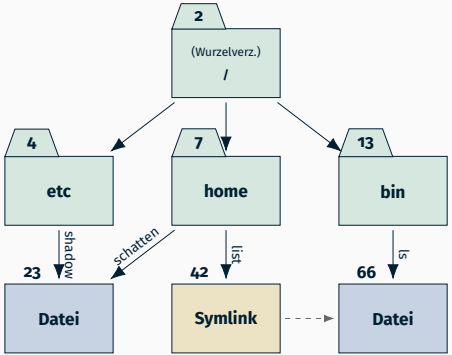
```
root:!:17497::
mon:*:17457::
bin:*:17457::
...
```

42

?

66

```
1010011100
0000111101
0100101000
0000000110
1101101100
0000111010
0111110100
1000101111
1110111101
0111011100
```



Beispiel Dateisystem (UNIX)

Wurzelverzeichnis

2

2	.
2	..
4	etc
7	home
13	bin

etc

4

4	.
2	..
23	shadow

home

7

7	.
2	..
23	schatten
42	list

bin

13

13	.
2	..
66	ls

23

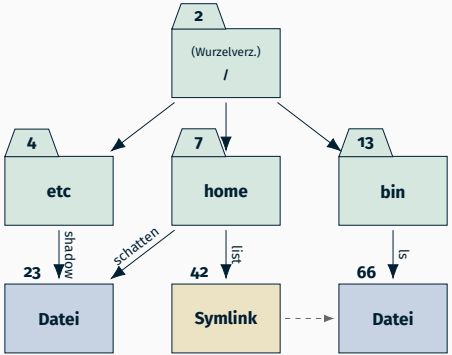
```
root:!:17497::
mon:*:17457::
bin:*:17457::
...
```

42

```
/bin/ls
```

66

```
1010011100
0000111101
0100101000
0000000110
1101101100
0000111010
0111110100
1000101111
1110111101
0111011100
```



Rechte auf Verzeichniseinträgen (UNIX)

- 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

Beispiel `ls` (UNIX)

- `ls` kann auch Inodes anzeigen (`-i`)

```
%> ls -oai /
total 140
  2  drwxr-xr-x  28 root  4096 Jun 24  2020 .
  2  drwxr-xr-x  28 root  4096 Jun 24  2020 ..
  4  drwxr-xr-x 180 root 12288 Jun  8 16:14 etc
  7  drwxr-xr-x   4 root  4096 Mär 15  2021 home
  13 drwxr-xr-x   2 root 12288 Jun  7 18:57 bin
  ...

%> ls -oai /home
total 2424
  7  drwxr-xr-x  28 root  4096 Jun 24 2020 .
  2  drwxr-xr-x  28 root  4096 Jun 24 2020 ..
  23 -rw-r-----   1 root  1479 Mär 16 2021 schatten
  42 lrwxrwxrwx   1 root     7 Jun 9  2021 list -> /bin/ls
  ...
```

Agenda

8.1 Aufbau eines Dateisystems

8.2 Dateisystem-Schnittstelle

8.3 Gelerntes anwenden

Dateiinformationen auslesen

- `stat(2)`/`lstat(2)` liefern Datei-Attribute aus 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 Dateiinformationen

Aufbau der Struktur `struct stat`

- Dateiattribute:

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    /* [...] */
    off_t     st_size;        /* Total size, in bytes */
    /* [...] */
};
```

Aufbau der Struktur `struct stat`

- Dateiattribute:

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    /* [...] */
    off_t     st_size;        /* Total size, in bytes */
    /* [...] */
};
```

Dateityp bestimmen

```
struct stat buf;
if(lstat("foo", &buf) == -1) { /* FB */ }

if(S_ISDIR(buf.st_mode)) {
    // directory
} else if(S_ISREG(buf.st_mode)) {
    // regular file
}
```

`S_ISREG` regular file?

`S_ISDIR` directory?

`S_ISBLK` block device?

Details → `inode(7)`

Dateiinhalte lesen/schreiben

- Relevante Funktionen:
  - `fopen(3)` öffnet eine (reguläre) Datei
  - `fclose(3)` schließt eine geöffnete Datei
  - `fflush(3)` leert Benutzer-Zwischenspeicher (nicht Kernel-Zwischenspeicher)
- Funktions-Prototypen

```
FILE *fopen(const char *pathname, const char *mode);

int fclose(FILE *stream);
```

- Benutzung
  - `FILE`-Zeiger können mit z.B. `fgets(3)`/`fputs(3)` verwendet werden
  - `mode` regelt Verwendungsmöglichkeiten
    - `r` nur lesend
    - `w` nur schreibend (bestehende Inhalte löschen)
    - `r+` lesend und schreibend
    - `a` anhängend
    - …

Verzeichnisinhalte auslesen

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

- DIR-Struktur ist ein Iterator und speichert jeweils aktuelle Position

`opendir`

Verzeichnisinhalte auslesen

```
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

Verzeichnisinhalte auslesen

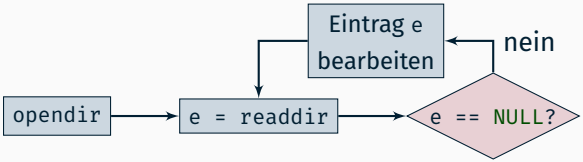
```
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

- `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

Verzeichnisinhalte auslesen

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

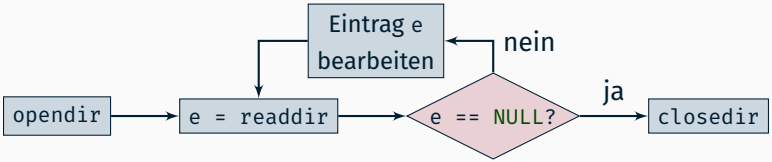
- `DIR`-Struktur ist ein Iterator und speichert jeweils aktuelle Position
- `readdir(3)` liefert einen Verzeichniseintrag und setzt den `DIR`-Iterator auf den Folgeeintrag
  - Rückgabewert `NULL` im Fehlerfall oder wenn EOF erreicht wurde
    - bei EOF bleibt **`errno`** unverändert, im Fehlerfall wird **`errno`** entsprechend gesetzt



Verzeichnisinhalte auslesen

```
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



Aufbau der Struktur `struct dirent`

- 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 **nicht** verwenden werden

Diskussion der Schnittstelle von `readdir(3)`

- 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

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
  - `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)

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
  - `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

Agenda

8.1 Aufbau eines Dateisystems

8.2 Dateisystem-Schnittstelle

8.3 Gelerntes anwenden

Aktive Mitarbeit!

„Aufgabenstellung“

- Ausgabe aller Dateinamen von symbolischen Verknüpfungen im aktuellen Verzeichnis