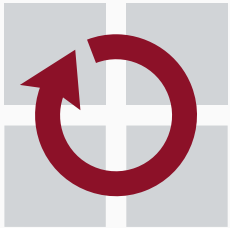
Übungen zu Systemprogrammierung 1

Ü6 – Dateisystem

Sommersemester 2025

Luis Gerhorst, Thomas Preisner, Tobias Häberlein, Jürgen Kleinöder

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



Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

7.1 Aufbau eines Dateisystems

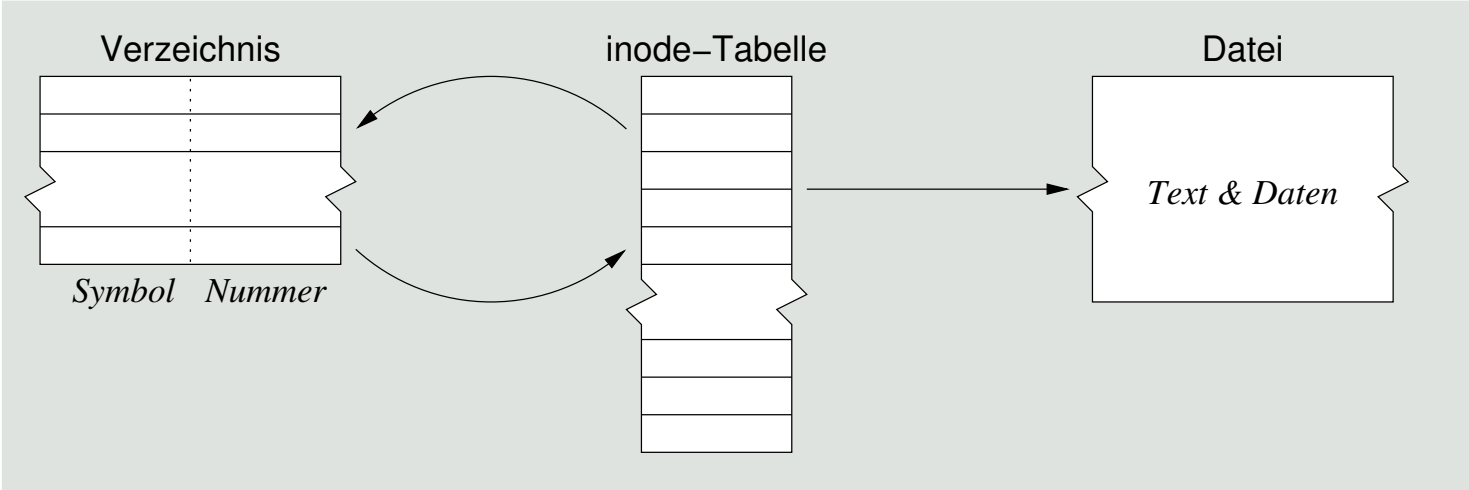
7.2 Dateisystem-Schnittstelle

7.3 Wildcards

7.4 Gelerntes anwenden

Agenda

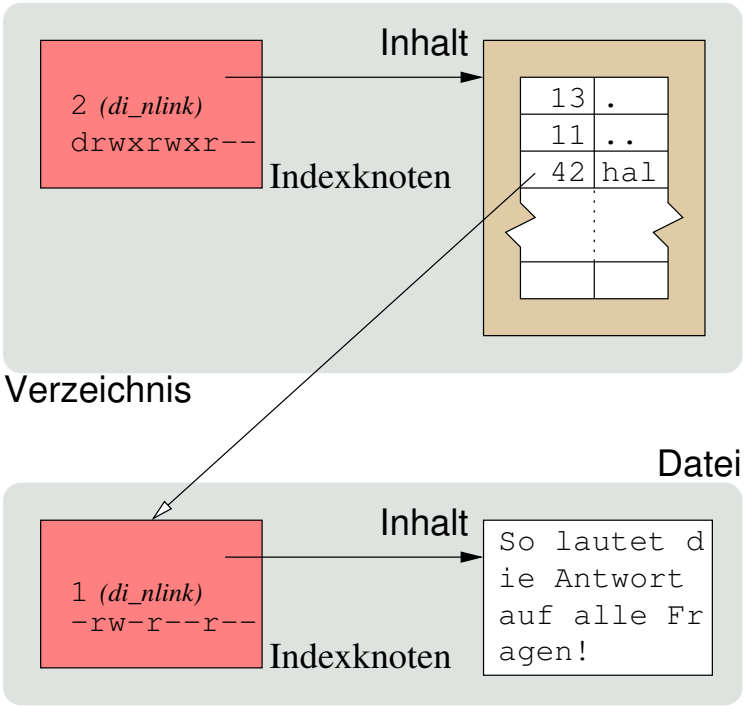
Zusammenhang der Datenstrukturen im Namensraum



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

Verzeichniseintrag II

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



- das selbst einen Namen hat, der auf einen *inode* verweist
- 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 Elternverzeichnis („*dot dot*“, 11) geben wenigstens zwei Verweise in einem Verzeichnis
  - auch wenn das Verzeichnis selbst sonst keine weiteren Namen enthält

Rechte auf Verzeichniseinträgen

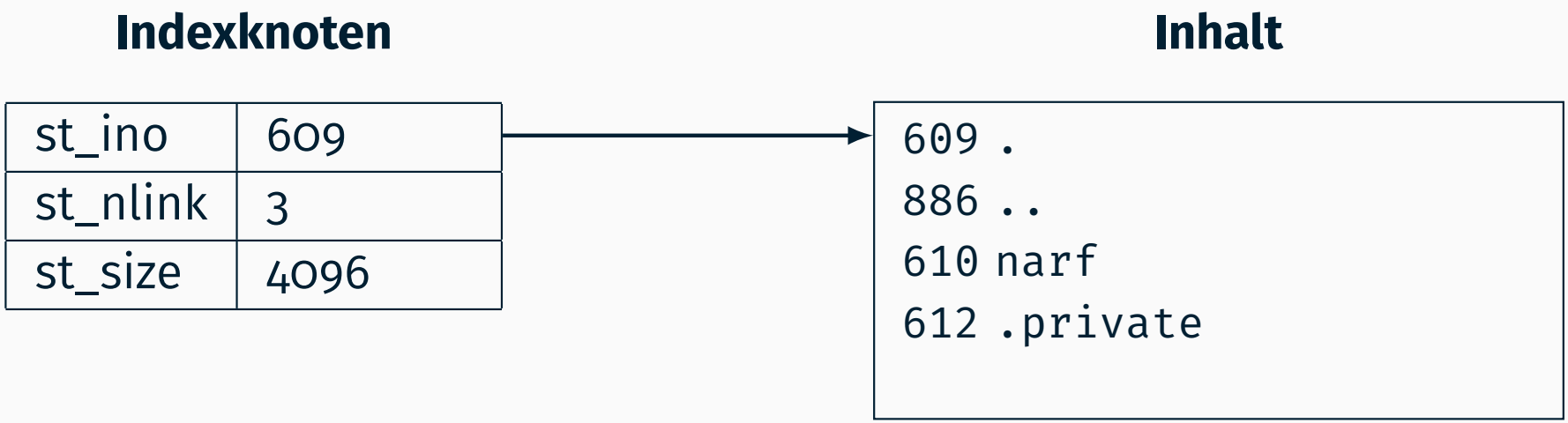
- UNIX sieht folgende Zugriffsrechte vor (davor die Darstellung des jeweiligen Rechts bei der Ausgabe des `ls`-Kommandos)
  - `r` lesen (getrennt für User, Group und Others einstellbar)
  - `w` schreiben (analog)
  - `x` ausführen (bei regulären Dateien) bzw. Durchgriffsrecht (bei Verzeichnissen)
  - `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 -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
```

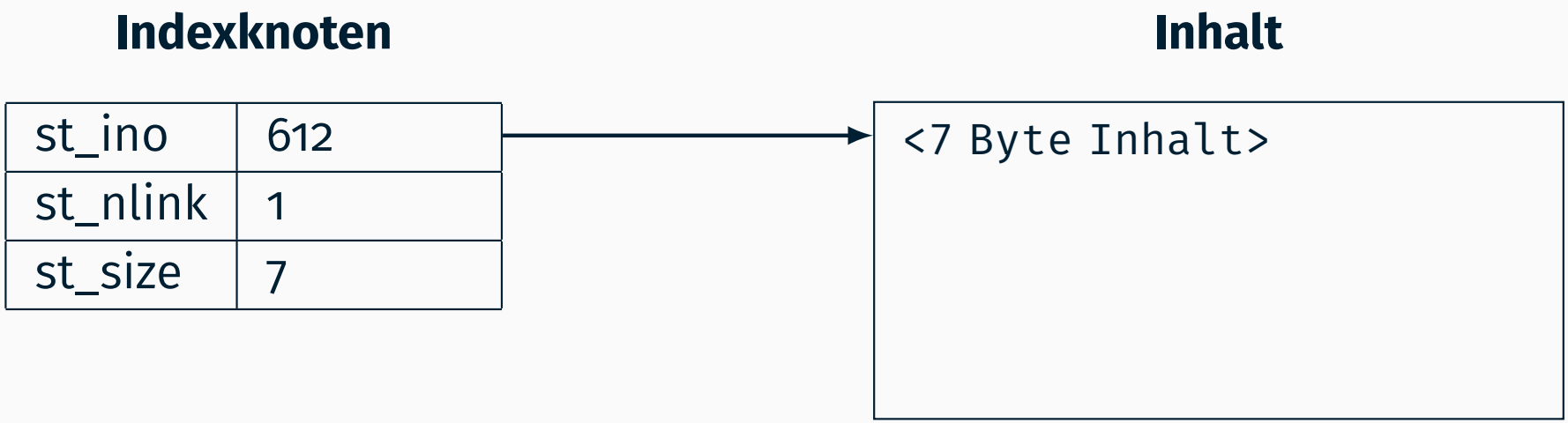
Beispiel

```
> 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
```



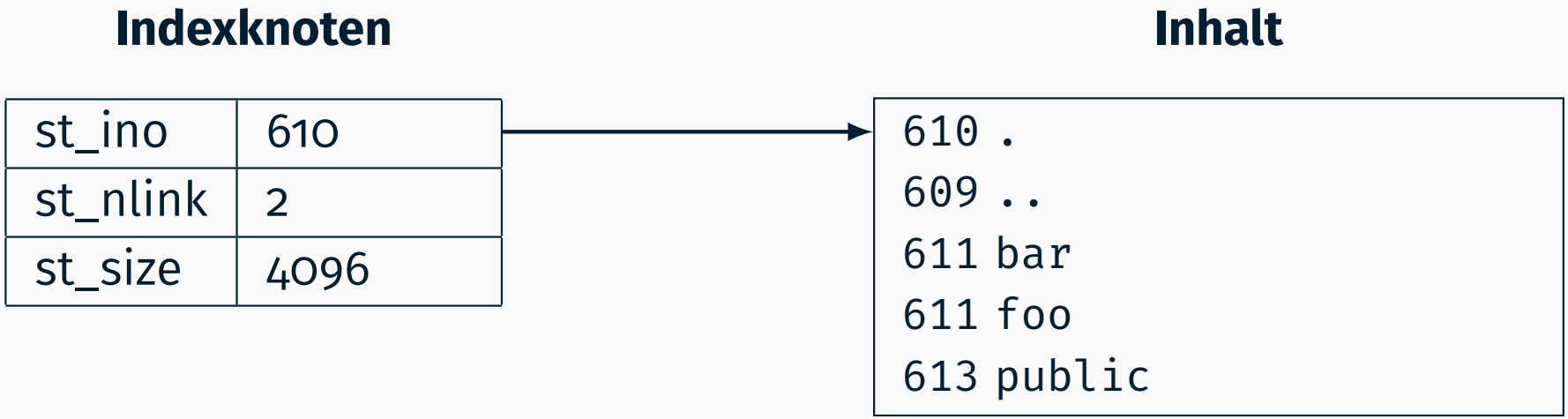
Beispiel

```
> 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
```



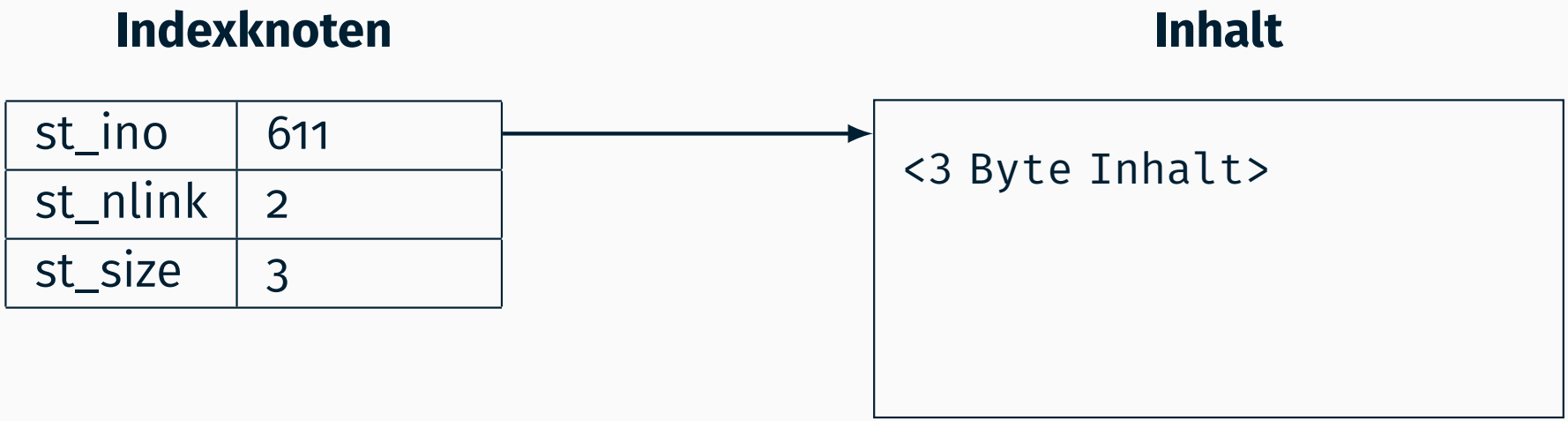
Beispiel

```
> 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
```



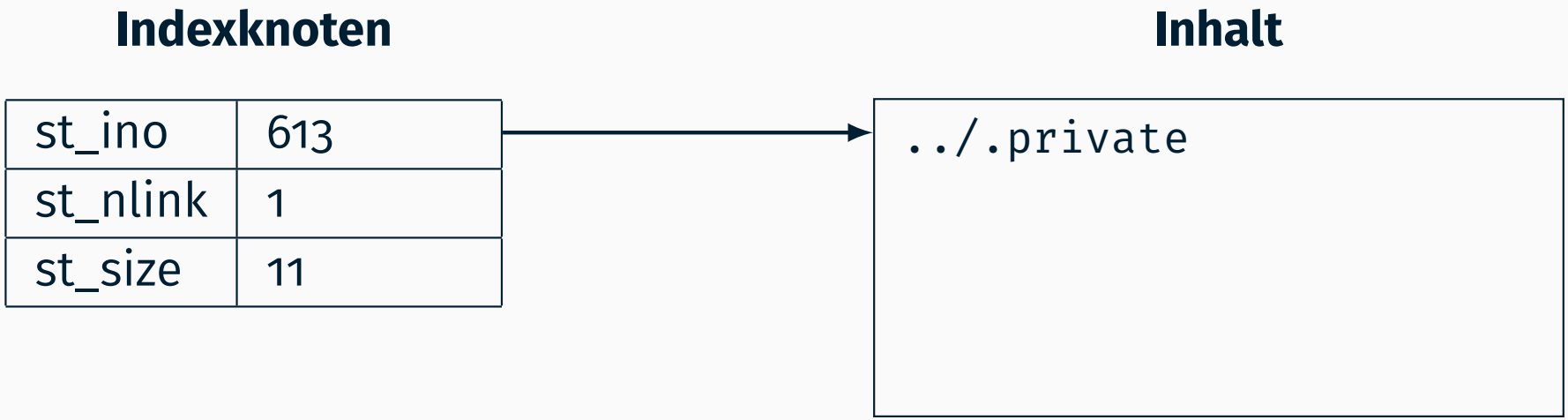
Beispiel

```
> 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
```



Beispiel

```
> 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
```



Agenda

Dateiinformationen auslesen

- `stat(3p)`/`lstat(3p)` liefern Datei-Attribute aus dem Inode
- Unterschiedliches Verhalten bei Symlinks:
  - `stat(3p)` folgt Symlinks (rekursiv) und liefert Informationen übers Ziel
  - `lstat(3p)` liefert Informationen über den Symlink selber
- Funktions-Prototypen

```c
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

Verzeichnisinhalte auslesen

```c
int scandir(
  const char *dir,
  struct dirent ***namelist,
  int (*sel)(const struct dirent *),
  int (*compar)(const struct dirent **, const struct dirent **)
);
```

- `scandir(3p)` speichert Einträge aus dem Verzeichnis `dir` in den Ausgabeparameter `namelist`
- Falls nur bestimmte Einträge gewünscht sind, kann die Auswahl mit dem Funktionsparameter `sel` eingeschränkt werden
  - Funktion wird für jeden Verzeichniseintrag aufgerufen
  - Gibt die Funktion `0` zurück, dann wird der Eintrag nicht in `namelist` aufgenommen.
- Die Verzeichniseinträge werden sortiert in `namelist` abgelegt. Dafür wird die Funktion `compar` aufgerufen (vgl. `qsort(3p)`)
  - Meist Verwendung von `alphasort(3p)`

Verzeichnisinhalte auflisten (Fortsetzung)

```
int scandir(
  const char *dir,
  struct dirent ***namelist,
  int (*sel)(const struct dirent *),
  int (*compar)(const struct dirent **, const struct dirent **)
);
```

- Im Erfolgsfall wird die Anzahl der Elemente in `namelist` zurückgegeben
- Der Speicher für das Array `namelist` und dessen Elemente wird von `scandir(3p)` selbst allokiert
  - ⇒ Freigabe mit `free(3p)` für jedes Element und `namelist` selbst nötig!
- Funktion kann aus diversen Gründen fehlschlagen (Manpage).
  - Rückgabewert `-1` im Fehlerfall
  - ⇒ Prüfen der **`errno`** und passende Fehlerbehandlung!

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 in SP **nicht** verwenden werden

Pfadnamen in POSIX-Systemen

- Arten
  - Absolute Pfade: beginnen mit / (Slash)
  - Relative Pfade: relativ zum aktuellen Arbeitsverzeichnis
- Mehrere direkt aufeinander folgende Slash-Zeichen werden als **ein einzelnes** Slash gewertet
  - Z.B. `/usr///bin//gcc` $\equiv$ `/usr/bin/gcc`
- **Ausnahme**: **Genau zwei** führende Slash-Zeichen werden implementierungsabhängig interpretiert
  - Z.B. `//var/log` $\not\equiv$ `/var/log`
  - Aber: `///var/log` $\equiv$ `/var/log`

Hintergrund

Ältere UNIX-Systeme nutzten diese Funktionen, um Teile des Netzwerks in das Dateisystem einzubinden.
Z.B. `//hostname/path/on/other/computer`
Der POSIX-Layer für Windows (Cygwin) verwendet dies bis heute.

Agenda

Wildcards

- ... 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

Fun with Wildcards

	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	

Wildcards auswerten

- … mit der Funktion `fnmatch(3p)`

```
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

Aktive Mitarbeit!

„Aufgabenstellung“

- Ausgabe aller Dateinamen von symbolischen Verknüpfungen im aktuellen Verzeichnis
- Ausgabe aller Übereinstimmungen von beliebig vielen Zeichenketten auf eine Wildcard, wobei beides als Kommandozeilenargumente übergeben werden sollen (`./matcher "*.txt" 1.txt 2.log 3`)