Systemnahe Programmierung in C (SPiC)

26 Dateisysteme – UNIX

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2022

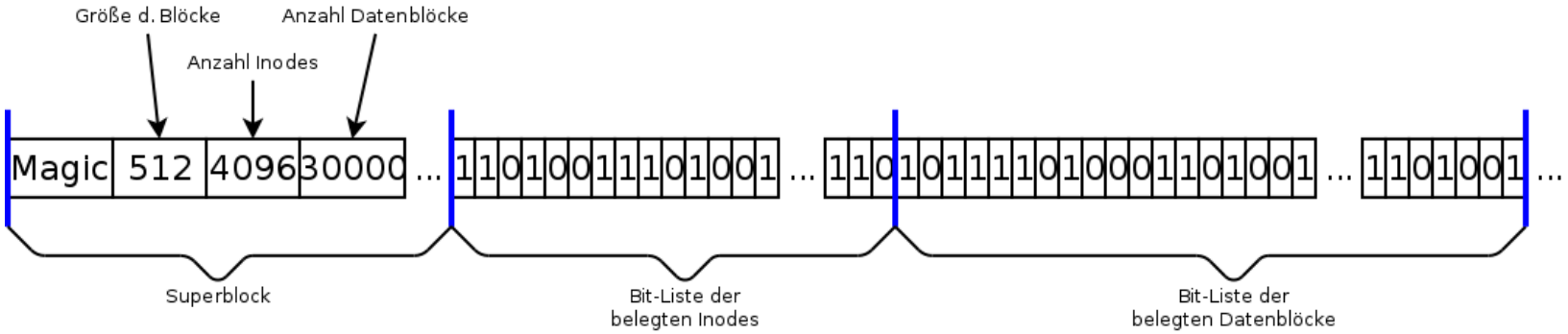
http://sys.cs.fau.de/lehre/SS22/spic

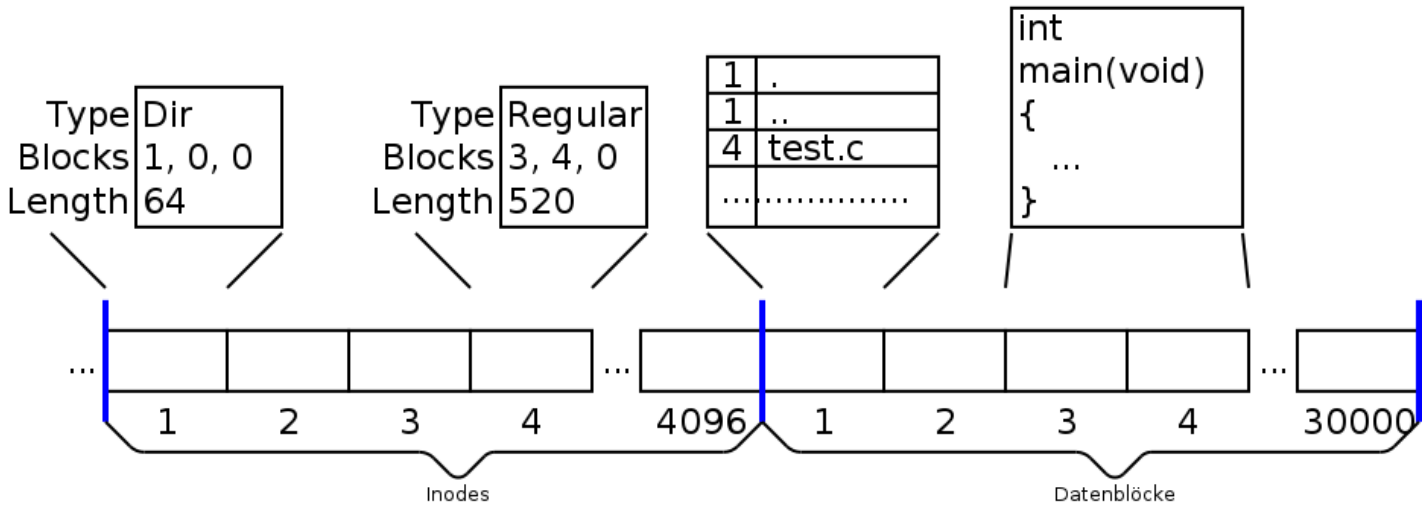
Dateisystem am Beispiel Linux/UNIX

- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbar
- Dateiattribute
  - das Betriebssystem verwaltet zu jeder Datei eine Reihe von Attributen (Rechte, Größe, Zugriffszeiten, Datenblöcke, ...)
  - die Attribute werden in einer speziellen Verwaltungsstruktur, dem *Dateikopf*, gespeichert
    – Linux/UNIX: *Inode*
    – Windows NTFS: *Master File Table*-Eintrag
- Namensraum
  - flacher Namensraum: Inodes sind einfach durchnummeriert
  - hierarchischer Namensraum: Verzeichnisstruktur bildet Datei- und Pfadnamen in einem Dateibaum auf Inode-Nummern ab

Dateisystem-Struktur

- Struktur auf Medium (vereinfacht)

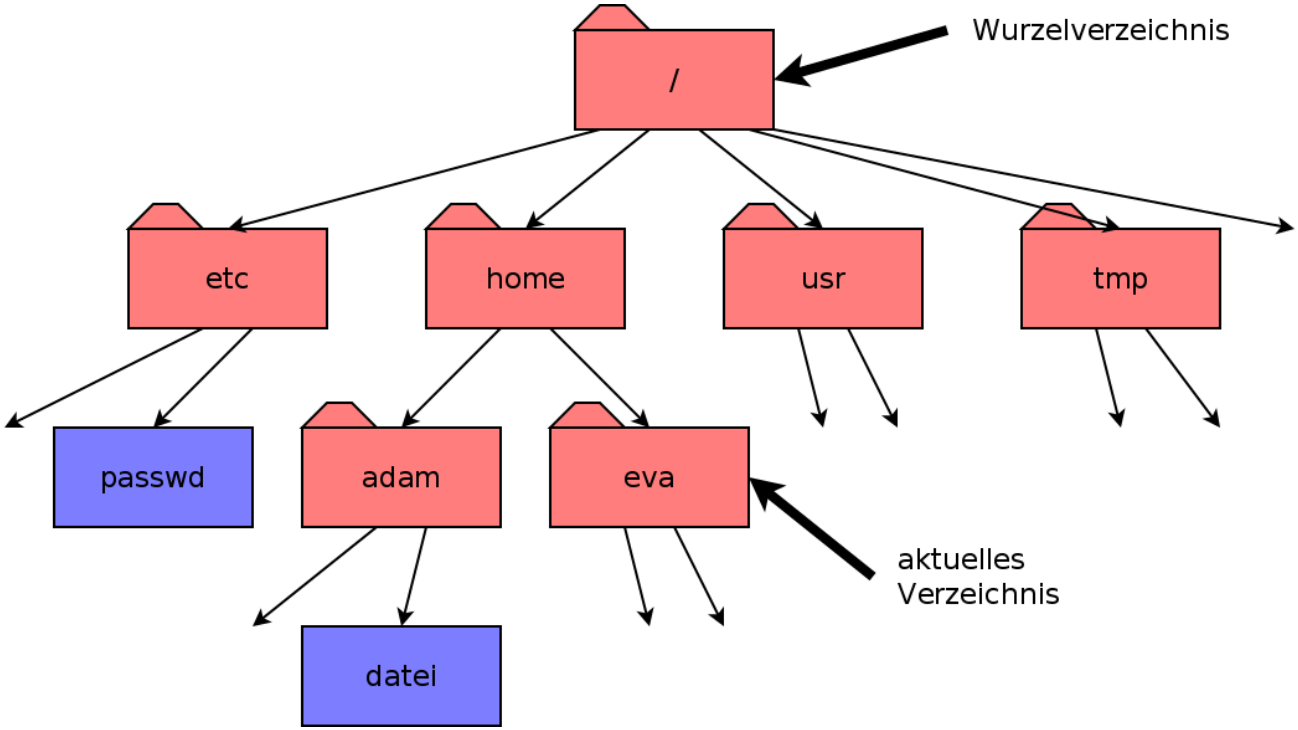




- `mkfs` legt leere Struktur an; `fsck` überprüft Struktur

Pfadnamen

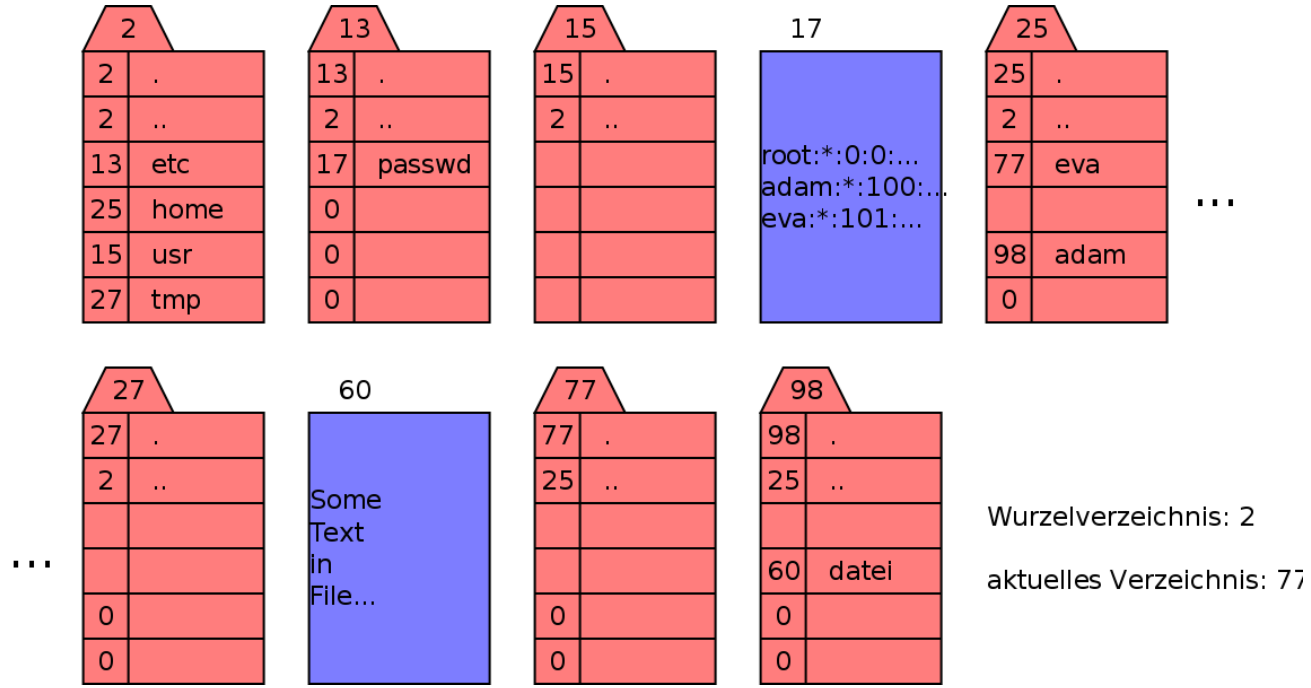
- Baumstruktur



- Pfade
  - z.B. `/home/adam/datei`, `/tmp`, `../adam/datei`
  - `/` ist Trennsymbol (*Slash*)
  - beginnender `/` ist Wurzelverzeichnis; sonst Beginn implizit mit dem aktuellen Verzeichnis

Pfadnamen (2)

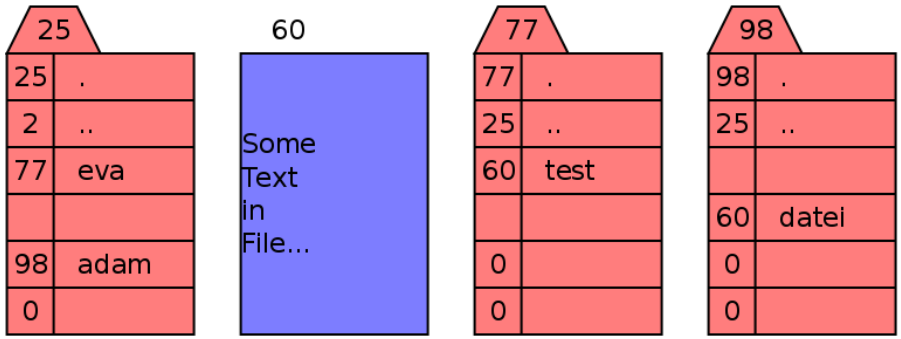
- eigentliche „Baumstruktur”



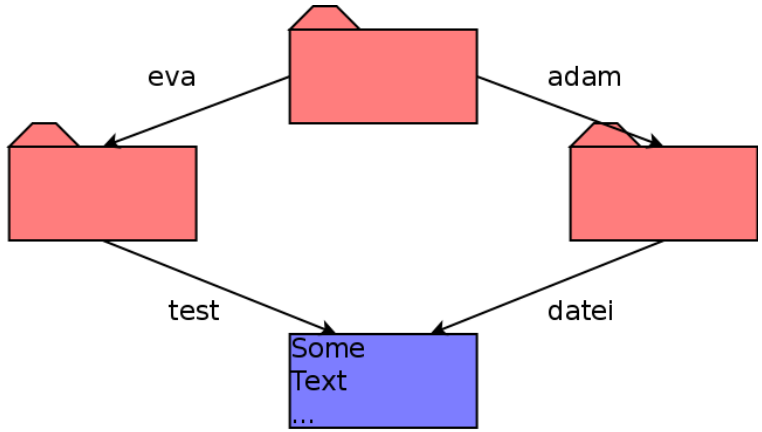
- Beispiel Pfadauflösung „`../adam/datei`”:
  - 77 + „`../adam/datei`” $\rightsquigarrow$ 25 + „`adam/datei`”
  - 25 + „`adam/datei`” $\rightsquigarrow$ 98 + „`datei`”
  - 98 + „`datei`” $\rightsquigarrow$ 60

Pfadnamen (3)

- Es können mehrere Verweise (**Hard Links**) auf eine Datei existieren:



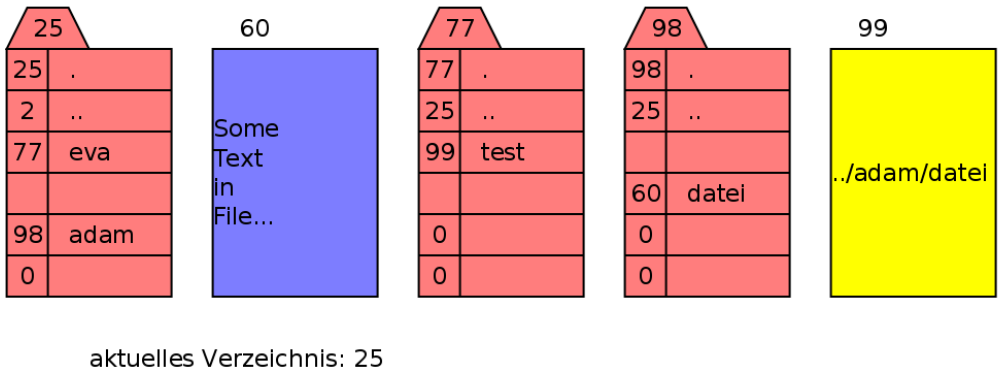
aktuelles Verzeichnis: 25

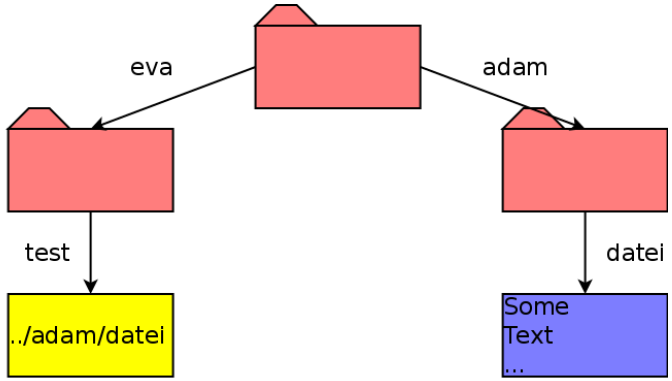


- Beispiel Pfadauflösung „`adam/datei`”:
  - 25 + „`adam/datei`” $\rightsquigarrow$ 98 + „`datei`”
  - 98 + „`datei`” $\rightsquigarrow$ 60
- Beispiel Pfadauflösung „`eva/test`”:
  - 25 + „`eva/test`” $\rightsquigarrow$ 77 + „`test`”
  - 77 + „`test`” $\rightsquigarrow$ 60
- Datei wird gelöscht, wenn keine Verweise auf sie mehr existieren

Pfadnamen (4)

- Es können mehrere symbolische Verweise (**Symbolic Links**) auf eine Datei oder ein Verzeichnis existieren:





- Beispiel Pfadauflösung „`eva/test`”:
  - 25 + „`eva/test`” $\rightsquigarrow$ 77 + „`test`”
  - 77 + „`test`” $\rightsquigarrow$ 99 $\rightsquigarrow$ 77 + „`../adam/datei`”
  - 77 + „`../adam/datei`” $\rightsquigarrow$ 25 + „`adam/datei`”
  - 25 + „`adam/datei`” $\rightsquigarrow$ 98 + „`datei`”
  - 98 + „`datei`” $\rightsquigarrow$ 60
- Symbolischer Name kann auch bestehen, wenn Datei oder Verzeichnis noch nicht bzw. nicht mehr existiert.

Eigentümer und Rechte

- Eigentümer
  - Jeder Eigentümer wird durch eindeutige Nummer (UID) repräsentiert
  - Ein Benutzer kann einer oder mehreren Benutzergruppen angehören, die jeweils durch eine eindeutige Nummer (GID) repräsentiert werden
  - Eine Datei oder ein Verzeichnis ist genau einem Benutzer und einer Gruppe zugeordnet
- Rechte auf Dateien
  - Lesen, Schreiben, Ausführen (nur vom Eigentümer änderbar)
  - Einzeln für den Eigentümer, für Angehörige der Gruppe und für alle anderen einstellbar
- Rechte auf Verzeichnissen
  - Lesen, Schreiben (Anlegen und Löschen von Dateien/Verzeichnissen), Durchgangsrecht
  - Schreibrecht ist einschränkbar auf eigene Dateien

Inodes

- Attribute (Zugriffsrechte, Eigentümer, usw.) einer Datei, eines Verzeichnisses werden in **Inodes** gespeichert (vereinfacht):

```
int st_mode;        /* Typ und Zugriffrechte */
int st_nlink;       /* Anzahl der Hard Links */
int st_uid;         /* Eigentuemer */
int st_gid;         /* Gruppe */
long st_size;       /* Laenge der Datei in Bytes */
int st_block[...]; /* Liste der (indirekten) Bloecke */
time_t st_atime;    /* Letzter Lesezeitpunkt */
time_t st_mtime;    /* Letzter Modifikationszeitpunkt */
time_t st_ctime;    /* Letzte Aenderung an Attributen */
```

- Jede Inode hat eine Nummer und einen Speicherort (Platte/Partition):

```
int st_ino;         /* Inode-Nummer */
int st_dev;         /* Platte/Partition-Nummer */
```


Programmierschnittstelle für Inodes

- `stat`, `lstat` liefern Dateiattribute aus Inodes
- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Argumente:
  - `path`: Pfadname
  - `buf`: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert
  - 0, wenn OK
  - -1, wenn Fehler (`errno`-Variable enthält Fehlernummer)
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerbehandlung...! */
printf("Inode-Nummer: %d\n", buf.st_ino);
```

Programmierschnittstelle für Verzeichnisse

- Verzeichnisse, Links verwalten
  - Erzeugen (eines leeren Verzeichnisses)

```
int mkdir(const char *path, mode_t mode);
```

  - Löschen (eines leeren Verzeichnisses)

```
int rmdir(const char *path);
```

  - Hard Link anlegen

```
int link(const char *existing, const char *new);
```

  - Symbolischen Link anlegen

```
int symlink(const char *existing, const char *new);
```

  - Link löschen (und damit ggf. auch Datei)

```
int unlink(const char *path);
```

  - Symbolischen Link auslesen

```
int readlink(const char *path, char *buf, int size);
```

Programmierschnittstelle für Verzeichnisse (2)

- Verzeichnisse lesen (Schnittstelle des Linux-Kerns)
  - open(2), getdents(2), close(2)
  - Linux-spezifisch und damit nicht portabel
- Verzeichnisse lesen (Schnittstelle der C-Bibliothek)
  - Verzeichnis öffnen

```
DIR *opendir(const char *path);
```

  - einen Eintrag lesen

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

  - Verzeichnis schließen

```
int closedir(DIR *dirp);
```

  - Struktur `struct dirent` (vereinfacht)

```
struct dirent {
    int d_ino;                  /* Inode-Nummer */
    char d_name[NAME_MAX + 1]; /* Name */
};
```


Verzeichnisse (3): `opendir/closedir`

- Funktionsschnittstelle:

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

DIR *opendir(const char *path);
int closedir(DIR *dirp);
```

- Argument von `opendir`:
  - `path`: Verzeichnisname
- Rückgabewert von `opendir`:
  - Zeiger auf Datenstruktur vom Typ `DIR`, wenn OK
  - `NULL`, wenn Fehler (`errno`-Variable enthält Fehlernummer)

Verzeichnisse (4): `readdir`

- Funktionsschnittstelle:

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

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

- Argument:
  - `dirp`: Zeiger auf `DIR`-Datenstruktur
- Rückgabewert:
  - Zeiger auf Datenstruktur vom Typ `struct dirent`, wenn OK
  - NULL, wenn Verzeichnis zu Ende gelesen wurde (`errno`-Variable nicht verändert)
  - NULL, wenn Fehler aufgetreten ist (`errno`-Variable enthält Fehlercode)
- Hinweis: Der Speicher für `struct dirent` wird u.U. beim nächsten `readdir`-Aufruf überschrieben!

Verzeichnisse (5): Beispiel

- Ausgabe der Dateinamen im aktuellen Verzeichnis:

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

DIR *dirp;
struct dirent *de;
int ret;

dirp = opendir(".");                    // akt. Verz. oeffnen
if (dirp == NULL) ...                   // Fehler

while (1) {
  errno = 0;
  de = readdir(dirp);                   // Eintrag lesen
  if (de == NULL && errno != 0) ...     // Fehler
  if (de == NULL) break;                // Ende erreicht

  printf("%s\n", de->d_name);
}

ret = closedir(dirp);                   // Verz. schliessen
if (ret < 0) ...                        // Fehler
```

Programmierschnittstelle für Dateien

- Funktionsschnittstelle:

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

int open(const char *path, int flags, ...);

int close(int fd);

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```


Dateien (2): Beispiel

- Kopierprogramm

```
#include <fcntl.h>

int ret;

int src_fd = open("src", O_RDONLY);
if (src_fd < 0) ...           // Fehler
int dst_fd = open("dst", O_CREAT | O_TRUNC | O_WRONLY, 0777);
if (dst_fd < 0) ...           // Fehler

while (1) {
  char buf[1024];
  len = read(src_fd, buf, sizeof(buf));
  if (len < 0) ...            // Fehler
  if (len == 0) break;
  ret = write(dst_fd, buf, len);
  if (ret < 0) ...            // Fehler
}

ret = close(dst_fd);
if (ret < 0) ...              // Fehler
ret = close(src_fd);
if (ret < 0) ...              // Fehler
```

Dateien (3)

- `write`-Aufruf muss
  - den File-Deskriptor überprüfen (Datei geöffnet, Datei beschreibbar?)
  - die Pufferadresse/-länge überprüfen
  - den/die zu beschreibenden Blöcke des Mediums ermitteln
  - den/die Blöcke vom Medium lesen (wenn nicht ganzer Block geschrieben wird)
  - die entsprechenden Bytes im gelesenen Block überschreiben
  - den/die Blöcke auf das Medium zurückübertragen
  - die Attribute anpassen (Datum letzte Modifikation, Länge der Datei)
  - und ist ein Betriebssystem-Aufruf
- => `write` ist eine zeitlich teure Operation (`read` entsprechend)!
- => Besser: viele Bytes (am Besten: Vielfache der Blockgröße) am Stück lesen/schreiben
- => `fopen`-, `fclose`-, `fread`-, `fwrite`-, `getchar`-, `putchar`-, `fscanf`-, `fprintf`-, ... -Funktionen aus der C-Bibliothek benutzen!

Spezialdateien

- Periphere Geräte (Platte, Drucker, CD, Terminal, Scanner, ...) werden als Spezialdateien repräsentiert (`/dev/sda`, `/dev/lp0`, `/dev/cdrom0`, `/dev/tty`, ...)
- in Inode steht
  - Typ:
    - Block-orientiertes Gerät (Platte, CD, DVD, SSD, ...)
    - Zeichen-orientiertes Gerät (Drucker, Terminal, Scanner, ...)
  - statt Blocknummern:
    - Major-Number: Typ des Gerätes (Platte, Drucker, ...)
    - Minor-Number: Nummer des Gerätes (3. Drucker, 5. Terminal, ...)
- Öffnen der Geräte schafft eine (evtl. exklusive) Verbindung zum Gerät, die durch Treiber hergestellt wird
- Geräte können dann mit `read`-, `write`- und `ioctl`-Operationen angesprochen werden

Spezialdateien (2): Beispiel

- Ausgabe auf Drucker

```
#include <linux/lp.h>
int fd, ret;

/* Verbindung zum Drucker 0 herstellen. */
fd = open("/dev/lp0", O_WRONLY);
if (fd < 0) ...

/* Druckerstatus abfragen. */
ret = ioctl(fd, LPGETSTATUS, &state);
if (ret < 0) ...
if (state & LP_POUTPA) {
  fprintf(stderr, "Out of paper!\n"); exit(1);
}

/* Auf Drucker schreiben. */
ret = write(fd, "Hallo, Drucker!\n\f", 17);
if (ret < 0) ...

/* Verbindung abbauen. */
ret = close(fd);
if (ret < 0) ...
```

Partitionen

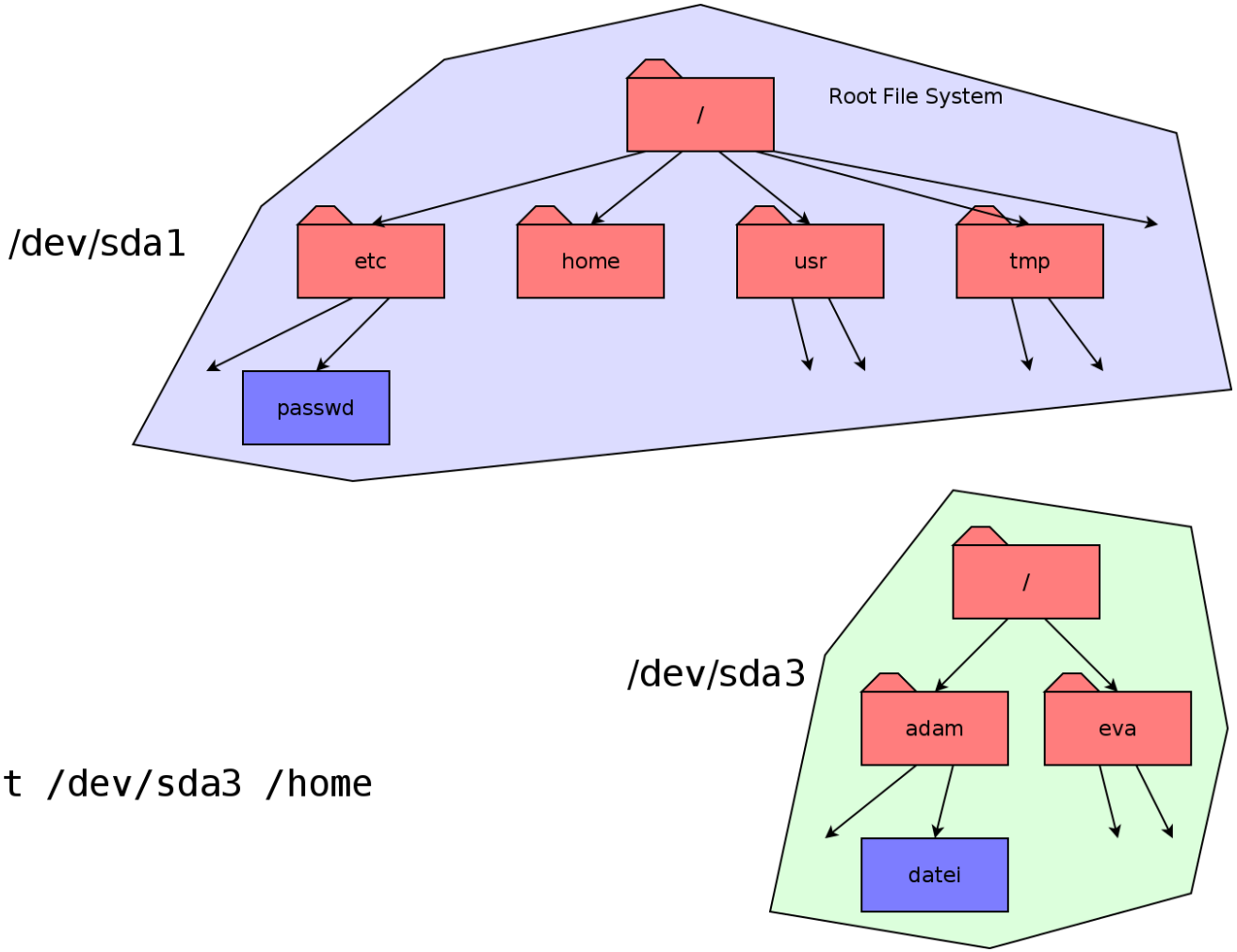
- jede Festplatte kann als Ganzes ein Dateisystem enthalten
  - Festplatte entspricht dann einer Partition
- jede Festplatte kann aber auch unterteilt werden in mehrere Partitionen
  - erster Block der Platte enthält Partitionstabelle
  - Partitionstabelle enthält Informationen
    - wieviele Partitionen existieren
    - wie groß die jeweiligen Partitionen sind
    - wo sie beginnen
- jede Partition
  - wird durch eine Spezialdatei repräsentiert; z.B.
    - `/dev/sda`, `/dev/sdb` (ganze Platte)
    - `/dev/sda1`, `/dev/sda2`, `/dev/sdb1` (Teile der jeweiligen Platte)
  - enthält eigenes Dateisystem

Partitionen (2)

- Bäume der Partitionen können zu einem homogenen Dateibaum zusammengesetzt werden (Grenzen für Anwender nicht sichtbar!)
  - „Montieren" von Dateibäumen (*mounting*)
- Ein ausgezeichnetes Dateisystem ist das *Root File System*, dessen Wurzelverzeichnis gleichzeitig das Wurzelverzeichnis des Gesamtsystems ist
  - Andere Dateisysteme können mit dem **`mount`**-Befehl in das bestehende System hineinmontiert werden bzw. mit dem **`umount`**-Befehl wieder entfernt werden.
  - Über das *Network File System* (NFS) können auch Verzeichnisse anderer Rechner in einen lokalen Dateibaum hineinmontiert werden => Grenzen zwischen Dateisystemen verschiedener Rechner werden unsichtbar

Montieren des Dateibaumes

- Beispiel




```
mount /dev/sda3 /home
```

Montieren des Dateibaumes (2)

- Nach Ausführung des Montierbefehls