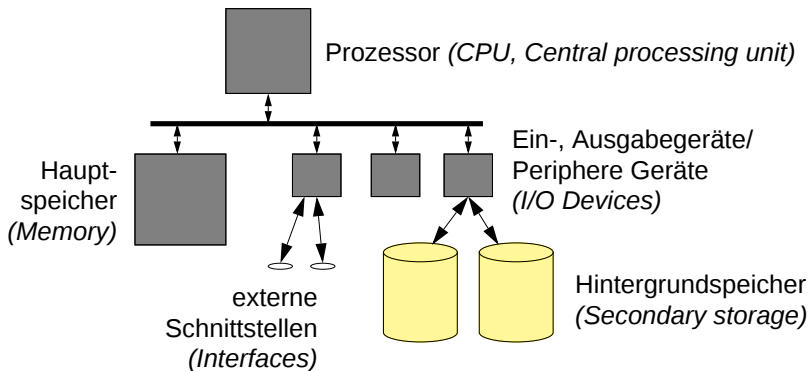


Allgemeine Konzepte

■ Einordnung



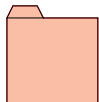
Allgemeine Konzepte (2)

- Dateisysteme speichern Daten und Programme persistent in Dateien
 - Betriebssystemabstraktion zur Nutzung von Hintergrundspeichern (z.B. Platten, CD-ROM, Bandlaufwerke)
 - Benutzer muss sich nicht um die Ansteuerungen verschiedener Speichermedien kümmern
 - einheitliche Sicht auf den Hintergrundspeicher
- Dateisysteme bestehen aus
 - Dateien (*Files*)
 - Katalogen (*Directories*)
 - Partitionen (*Partitions*)

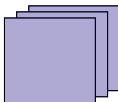


Allgemeine Konzepte (3)

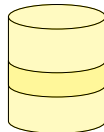
- Datei
 - speichert Daten oder Programme
- Katalog / Verzeichnis (*Directory*)
 - erlaubt Benennung der Dateien
 - enthält Zusatzinformationen zu Dateien
- Partitionen
 - eine Menge von Katalogen und deren Dateien
 - sie dienen zum physischen oder logischen Trennen von Dateimengen.



Katalog



Dateien



Partition



Überblick

- E-/A-Funktionalität nicht Teil der Programmiersprache
- Realisierung durch "normale" Funktionen
 - Bestandteil der Standard-Funktionsbibliothek
 - einfache Programmierschnittstelle
 - effizient
 - portabel
 - betriebssystemnah
- Funktionsumfang
 - Öffnen/Schließen von Dateien
 - Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
 - Formatierte Ein-/Ausgabe



Standard Ein-/Ausgabe

- Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:
 - **stdin** Standardeingabe
 - normalerweise mit der Tastatur verbunden, Umlenkung durch **<**
 - Dateiende (**EOF**) wird durch Eingabe von **CTRL - D** am Zeilenanfang signalisiert
 - **stdout** Standardausgabe
 - normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden, Umlenkung durch **>**
 - **stderr** Ausgabekanal für Fehlermeldungen
 - normalerweise ebenfalls mit Bildschirm verbunden
- automatische Pufferung
 - Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **NEWLINE**-Zeichen (**'\n'**) an das Programm übergeben!



Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen

- Zugriff auf Dateien

- Öffnen eines E/A-Kanals

- Funktion `fopen`

- Prototyp:

```
FILE *fopen(char *name, char *mode);
```

name Pfadname der zu öffnenden Datei

mode Art, wie die Datei geöffnet werden soll

"r" zum Lesen

"w" zum Schreiben

"a" append: Öffnen zum Schreiben am Dateiende

"rw" zum Lesen und Schreiben

- Ergebnis von **fopen**:

Zeiger auf einen Datentyp **FILE**, der einen Dateikanal beschreibt
im Fehlerfall wird ein **NULL**-Zeiger geliefert



Öffnen und Schließen von Dateien (2)

■ Beispiel:

```
#include <stdio.h>

int main() {
    FILE *eingabe;
    char dateiname[256];

    printf("Dateiname: ");
    scanf("%s\n", dateiname);

    if ((eingabe = fopen(dateiname, "r")) == NULL) {
        /* eingabe konnte nicht geöffnet werden */
        perror(dateiname); /* Fehlermeldung ausgeben */
        exit(1);           /* Programm abbrechen */
    }

    ... /* Programm kann jetzt von eingabe lesen */
    ... /* z. B. mit c = getc(eingabe) */
}
```

■ Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```

- schließt E/A-Kanal **fp**



Zeichenweise Lesen und Schreiben

■ Lesen eines einzelnen Zeichens

■ von der Standardeingabe

```
int getchar( )
```

➤ lesen das nächste Zeichen

➤ geben das gelesene Zeichen als **int**-Wert zurück

➤ geben bei Eingabe von **CTRL-D** bzw. am Ende der Datei **EOF** als Ergebnis zurück

■ von einem Dateikanal

```
int getc(FILE *fp )
```

■ Schreiben eines einzelnen Zeichens

■ auf die Standardausgabe

```
int putchar(int c)
```

➤ schreiben das im Parameter **c** übergeben Zeichen

➤ geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

■ auf einen Dateikanal

```
int putc(int c, FILE *fp )
```



Zeichenweise Lesen und Schreiben (2)

■ Beispiel: copy-Programm

```
#include <stdio.h>
```

Teil 1: Dateien öffnen

```
int main() {
    FILE *quelle;
    FILE *ziel;
    char quelldatei[256], zieldatei[256];
    int c;
    /* gerade kopiertes Zeichen */

    printf("Quelldatei und Zieldatei eingeben: ");
    scanf("%s %s\n", quelldatei, zieldatei);

    if ((quelle = fopen(quelldatei, "r")) == NULL) {
        perror(quelldatei); /* Fehlermeldung ausgeben */
        exit(1);           /* Programm abbrechen */
    }

    if ((ziel = fopen(zieldatei, "w")) == NULL) {
        perror(zieldatei); /* Fehlermeldung ausgeben */
        exit(1);           /* Programm abbrechen */
    }

    /* ... */
}
```



Zeichenweise Lesen und Schreiben (3)

... Beispiel: copy-Programm
— Fortsetzung

```
/* ... */                                     Teil 2: kopieren

while ( (c = getc(quelle)) != EOF ) {
    putc(c, ziel);
}

fclose(quelle);
fclose(ziel);
}
```



■ Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int printf(char *format, /* Parameter */ ... );  
int fprintf(FILE *fp, char *format, /* Parameter */ ... );  
int sprintf(char *s, char *format, /* Parameter */ ...);  
int snprintf(char *s, int n, char *format, /* Parameter */ ...);
```

Die statt ... angegebenen Parameter werden entsprechend der Angaben im **format**-String ausgegeben

- bei **printf** auf der Standardausgabe
- bei **fprintf** auf dem Dateikanal **fp**
(für **fp** kann auch **stdout** oder **stderr** eingesetzt werden)
- **sprintf** schreibt die Ausgabe in das **char**-Feld **s**
(achtet dabei aber nicht auf das Feldende
-> potentielle Sicherheitsprobleme!)
- **snprintf** arbeitet analog, schreibt aber maximal nur **n** Zeichen
(**n** sollte natürlich nicht größer als die Feldgröße sein)



- Zeichen im **format**-String können verschiedene Bedeutung haben
 - normale Zeichen: werden einfach auf die Ausgabe kopiert
 - Escape-Zeichen: z. B. `\n` oder `\t`, werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
 - Format-Anweisungen: beginnen mit %-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem **format**-String aufbereitet werden soll

- Format-Anweisungen

- %d, %i** **int** Parameter als Dezimalzahl ausgeben
- %f** **float** oder **double** Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
- %e** **float** oder **double** Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben
- %c** **char**-Parameter wird als einzelnes Zeichen ausgegeben
- %s** **char**-Feld wird ausgegeben, bis '`\0`' erreicht ist



■ Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int scanf(char *format, /* Parameter */ ...);  
int fscanf(FILE *fp, char *format, /* Parameter */ ...);  
int sscanf(char *s, const char *format, /* Parameter */ ...);
```

- Die Funktionen lesen Zeichen von **stdin** (**scanf**), **fp** (**fscanf**) bzw. aus dem **char**-Feld **s**.
- **format** gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen
- Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. **char**-Felder bei Format **%s**), in die die Resultate eingetragen werden
- relativ komplexe Funktionalität, hier nur Kurzüberblick
für Details siehe Manual-Seiten



Formatierte Eingabe — Eingabe-Daten

- *White space* (Space, Tabulator oder Newline \n) bildet jeweils die Grenze zwischen Daten, die interpretiert werden
 - *white space* wird in beliebiger Menge einfach überlesen
 - Ausnahme: bei Format-Anweisung **%c** wird auch *white space* eingelesen
- Alle anderen Daten in der Eingabe müssen zum **format**-String passen oder die Interpretation der Eingabe wird abgebrochen
 - wenn im format-String normale Zeichen angegeben sind, müssen diese exakt so in der Eingabe auftauchen
 - wenn im Format-String eine Format-Anweisung (%...) angegeben ist, muß in der Eingabe etwas hierauf passendes auftauchen
 - ➡ diese Daten werden dann in den entsprechenden Typ konvertiert und über den zugehörigen Zeiger-Parameter der Variablen zugewiesen
- Die **scanf**-Funktionen liefern als Ergebnis die Zahl der erfolgreich an die Parameter zugewiesenen Werte
- Detail siehe Manual-Seite (man scanf)



■ Datei

- einfache, unstrukturierte Folge von Bytes
- beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
- dynamisch erweiterbar

■ Katalog

- baumförmig strukturiert
 - Knoten des Baums sind Kataloge
 - Blätter des Baums sind Verweise auf Dateien
- jedem UNIX-Prozess ist zu jeder Zeit ein aktueller Katalog (*Current working directory*) zugeordnet

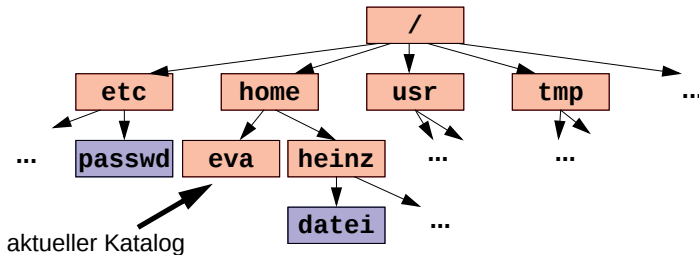
■ Partitionen

- jede Partition enthält einen eigenen Dateibaum
- Bäume der Partitionen werden durch "mounten" zu einem homogenen Dateibaum zusammengebaut (Grenzen für Anwender nicht sichtbar!)



Pfadnamen

Baumstruktur



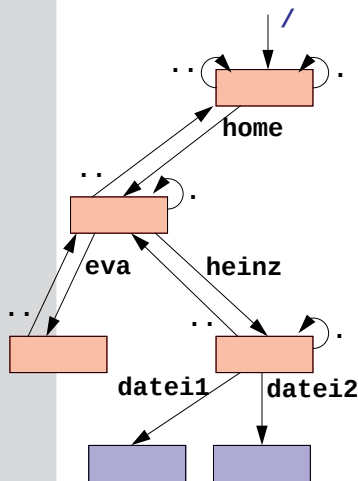
Pfade

- z.B. „`/home/heinz/datei`“, „`/tmp`“, „`../heinz/datei`“
- „`/`“ ist Trennsymbol (*Slash*); beginnender „`/`“ bezeichnet Wurzelkatalog; sonst Beginn implizit mit dem aktuellen Katalog



Pfadnamen (2)

■ Eigentliche Baumstruktur



▲ benannt sind nicht Dateien und Kataloge, sondern die Verbindungen (*Links*) zwischen ihnen

- Kataloge und Dateien können auf verschiedenen Pfaden erreichbar sein
z. B. **../heinz/datei1** und **/home/heinz/datei1**
- Jeder Katalog enthält
 - einen Verweis auf sich selbst (.) und
 - einen Verweis auf den darüberliegenden Katalog im Baum (..)
 - Verweise auf Dateien



■ Kataloge verwalten

■ Erzeugen

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

■ Löschen

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

■ Kataloge lesen (Schnittstelle der C-Bibliothek)

➤ Katalog öffnen:

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

➤ Katalogeinträge lesen:

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

➤ Katalog schließen:

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



Kataloge (2): opendir / closedir

■ Funktionsschnittstelle:

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

DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

■ Argument von opendir

■ **dirname**: Verzeichnisname

■ Rückgabewert: Zeiger auf Datenstruktur vom Typ **DIR** oder **NULL**



Kataloge (3): readdir

■ Funktionsschnittstelle:

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

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

■ Argumente

■ **dirp**: Zeiger auf **DIR**-Datenstruktur

■ Rückgabewert: Zeiger auf Datenstruktur vom Typ **struct dirent** oder **NULL** wenn fertig oder Fehler (**errno** vorher auf 0 setzen!)

■ Probleme: Der Speicher für **struct dirent** wird von der Funktion **readdir** beim nächsten Aufruf wieder verwendet!

- wenn Daten aus der Struktur (z. B. der Dateiname) länger benötigt werden, reicht es nicht, sich den zurückgegebenen Zeiger zu merken sondern es müssen die benötigten Daten kopiert werden



Kataloge (4): struct dirent

- Definition unter Linux (/usr/include/bits/dirent.h)

```
struct dirent {  
    __ino_t d_ino;  
    __off_t d_off;  
    unsigned short int d_reclen;  
    unsigned char d_type;  
    char d_name[256];  
};
```

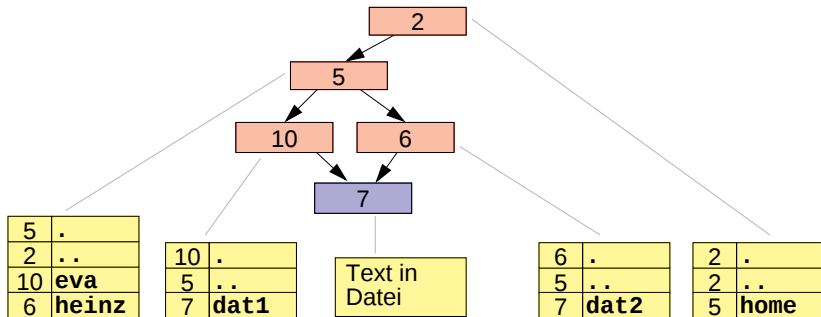
Programmierschnittstelle für Dateien

- siehe C-Ein/Ausgabe (Schnittstelle der C-Bibliothek)
- C-Funktionen (fopen, printf, scanf, getchar, fputs, fclose, ...) verbergen die "eigentliche" Systemschnittstelle und bieten mehr "Komfort"
 - Systemschnittstelle: open, close, read, write



Inodes

- Attribute (Zugriffsrechte, Eigentümer, etc.) einer Datei und Ortsinformation über ihren Inhalt werden in **Inodes** gehalten
 - Inodes werden pro Partition numeriert (*Inode number*)
- Kataloge enthalten lediglich Paare von Namen und Inode-Nummern
 - Kataloge bilden einen hierarchischen Namensraum über einem eigentlich flachen Namensraum (durchnummerierte Dateien)



Inodes (2)

- Inhalt eines Inode
 - Dateityp: Katalog, normale Datei, Spezialdatei (z.B. Gerät)
 - Eigentümer und Gruppe
 - Zugriffsrechte
 - Zugriffszeiten: letzte Änderung (*mtime*), letzter Zugriff (*atime*), letzte Änderung des Inodes (*ctime*)
 - Anzahl der Hard links auf den Inode
 - Dateigröße (in Bytes)
 - Adressen der Datenblöcke des Datei- oder Kataloginhalts



Inodes — Programmierschnittstelle: stat / lstat

- liefert Datei-Attribute aus dem Inode

- 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**: Dateiname
- **buf**: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden

- Rückgabewert: 0 wenn OK, -1 wenn Fehler

- Beispiel:

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

