Fakultät für Informatik

MNM TEAM
Munich Network Management Team

Institut für Informatik

LMU

Systempraktikum im Wintersemester 2009/2010 (LMU):
Vorlesung vom 19.10. – Foliensatz 1

Grundlagen von Betriebssystemen (T)
Einführung in die C-Programmierung (P)
Ein- und Ausgabe (E/A) (P)

Dr. Thomas Schaaf, Dr. Nils gentschen Felde

Grundlagen von Betriebssystemen

MNM TEAM

- Aufgaben eines Betriebssystems
  - Ressourcenverwaltung
  - Erweiterte Maschine → Abstraktion von Ressourcen
- (Minimale) Bestandteile eines Betriebssystems
  - Systemkern (Kernel)
    - Monolithischer Kernel
    - Mikrokernel
    - Makrokernel
  - Kommandozeileninterpreter (Shell)
  - (Weitere) Dienstprogramme (z.B. Compiler, Editor)

Grundlagen von Betriebssystemen

MNM TEAM

- Anforderungen ans Betriebssystem
  - Aufgaben des Betriebssystem-Kerns:
    - Prozessverwaltung → Prozesserzeugung, Scheduling
    - Speicherverwaltung → Partitionierung, Virtueller Speicher/Paging
    - Dateiverwaltung → Dateisysteme, Dateitabellen
    - Benutzerverwaltung
  - Sonstige Anforderungen:
    - Fehlerfreie Ausführung → Synchronisation, Deadlock-Behebung
    - Sicherheit → Authentifizierung
    - Einfachheit → maximale Komplexitätsreduktion
- Aufgaben der Systemprogrammierung
  - Programmierung des Betriebssystem-Kerns
  - Programmierung von Dienstprogrammen
  - Programmierung von Kommunikationssoftware

Grundlagen von Betriebssystemen

MNM TEAM

- Konzepte in der (System-)Programmierung
  - Sprachebenen:
    - Quelltext mit Makros
    - Programm in höherer Programmiersprache
    - Assemblerprogramm
    - Maschinenprogramm
  - Dienstprogramme im Zusammenhang mit der (System-)Programmierung:
    - Präprozessor/Makro-Übersetzer
    - Übersetzer (Compiler)
    - Assembler
    - Binder (Linker)

- Konzepte in der (System-)Programmierung (Forts.)
  - Schnittstelle (Interface):
    - Menge der an einer bestimmten Stelle zur Verfügung stehenden Funktionen
    - und deren Aufrufkonventionen
  - Dienstanforderungen an den Systemkern:
    - Systemaufruf (system call)
    - Bibliotheksfunktion (subroutine)

Bekannte Beispiele für Betriebssysteme

MNM TEAM

- Linux
  - Unix-ähnliches Betriebssystem
  - Erste Version: ca. 1991
  - Open Source
  - Linux-Distributionen (Auszug):
    - Debian
    - Fedora
    - Red Hat
    - Suse

- Microsoft Windows
  - Erste Version: 1985
  - Closed Source
  - Versionen (Auszug):
    - Vorläufer: MS DOS
    - Windows 3.1
    - Windows 95, 98, ME
    - Windows NT, 2000
    - Windows XP, Server 2003
    - Windows Vista, Windows 7

- Apple Mac OS
  - Erste Version: 1984
  - Closed Source
  - Aktuelle Version: Max OS X

Lizenzierung, Standardisierung

MNM TEAM

- GNU General Public License (GPL)
  - Teil des GNU-Projekts (Ziel: Freie Betriebssysteme)
  - Open Source Software unter GPL:
    - Freie Nutzung
    - Freies Kopieren/Weitergeben
    - Freies Anpassen
    - Freies Vertreiben veränderter Versionen
  - Linux-Kernel steht unter GPL
- Portable Operating System Interface (POSIX)
  - IEEE-Standard
  - Ein Betriebssystem ist POSIX-konform, wenn
    - es die in POSIX.1 definierten C-Headerdateien zur Verfügung stellt,
    - es die in POSIX.2 definierten Hilfsprogramme enthält,
    - es bestimmte C-Systemaufrufe (insb. aus den Bereichen E/A und Prozesskontrolle) unterstützt.

Einführung in die C-Programmierung

MNM TEAM

- C vs. C++
  - C++ ist eine Weiterentwicklung von C (ursprünglicher Name: "C mit Klassen")
  - C-Programme lassen sich mit einem C++-Compiler übersetzen, aber i.d.R. nicht umgekehrt
  - In C++ Objektorientierung, generische Programmierung
- C(++) vs. Java
  - C ist wesentlich "systemnaher"
  - Mehr Freiheiten, aber auch mehr "Verantwortung" beim Programmierer, oft fehleranfälliger
  - erlaubt z.B. Einbettung von Assembler-Code
  - Programmieren mit Speicher-Referenzen (Zeigern)
  - C(++)-Programme werden in Maschinen-ausführbaren Code (nicht in Bytecode für eine virtuelle Maschine) übersetzt

Einführung in die C-Programmierung

MNM TEAM

- Erstes Programm: "Hello World" (hello.c)

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

- Übersetzung:

```
gcc hello.c -o helloworld
```

- Ausführung:

```
./helloworld
```

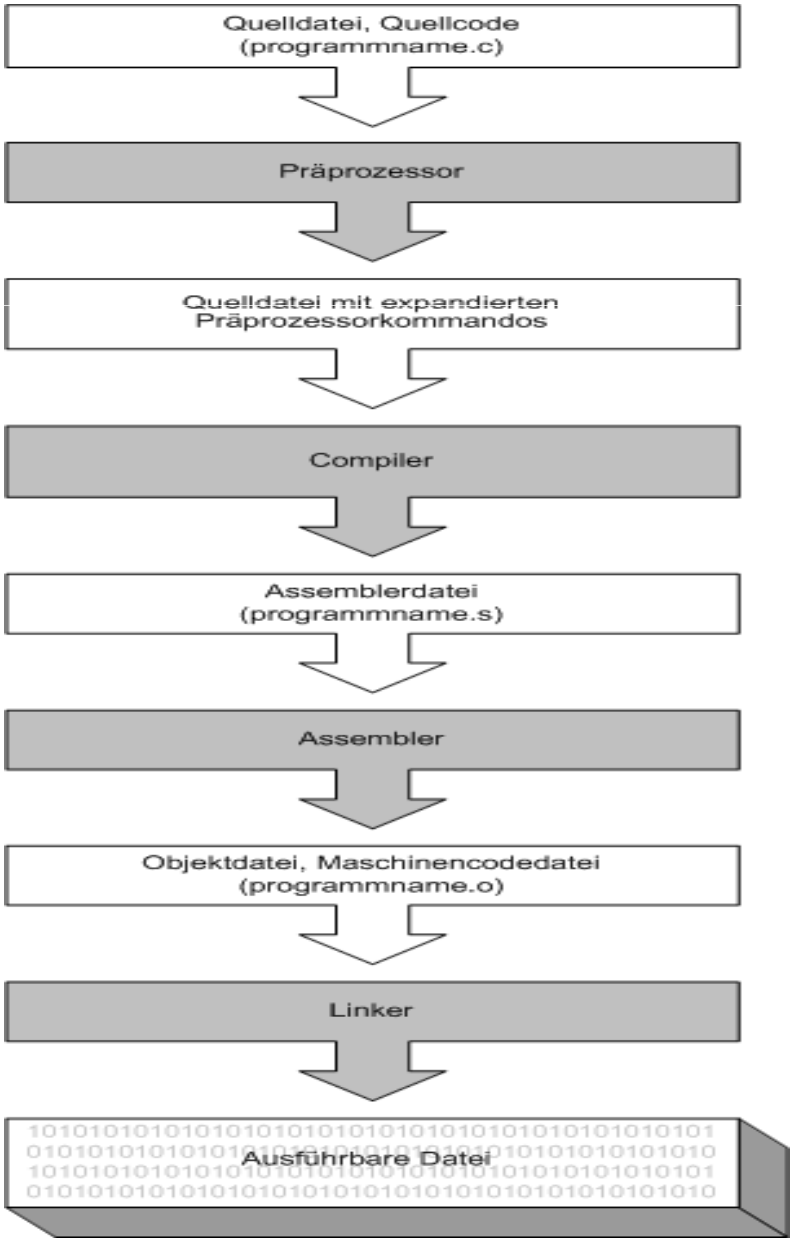
- Ausgabe:

```
Hello World!
```

- GNU C-Compiler
  - Aufruf: `gcc`
  - Hilfe/Beschreibung der Aufrufparameter (Flags): `man gcc`
  - Beispiele für Aufrufparameter:
    - `-o`: Name der erzeugten Programmdatei
    - `-Wall`: Alle Warnungen anzeigen
    - `-g`: Anhängen der Symboltabelle (ermöglicht Debugging)
  - Der gcc-Aufruf beinhaltet implizite Aufrufe von:
    - Präprozessor (Makro-Übersetzer): `cpp`
    - Eigentlicher GNU C-Compiler
    - Assembler
    - Linker
  - Nicht alle Phasen müssen ausgeführt werden:
    - Beispiel: `gcc -c`

      → Linker wird nicht aufgerufen, Ausgabe besteht aus den Objektdateien (*.o), die vom Assembler erzeugt wurden

- Vom Quellcode zum ausführbaren Programm



- Einfache Konsolenein-/ausgabe
  - Ausgabe:

```
printf("Ausgabetext");
```

  - Umwandlungszeichen (conversion modifiers):
    - Zeichenkette (String): %s
    - Einzelnes Zeichen (Character): %c
    - Ganze Zahl (Integer): %i
    - Gleitkommazahl (Float): %f
    - Oktalwert: %o
    - Hexadezimalwert: %x
    - Pointer (Zeiger/Adresse): %p
    - …
  - Einlesen von der Konsole:

```
int n; scanf("%i", &n);
```

- Beispiel: Ein-/Ausgabe
  - Was liefert dieses Programm (einausgabe.c)?

```
#include <stdio.h>
int main() {
    char z = 'K';
    char s[] = "Beliebiger Text\n";

    printf(s);
    printf("%c\n", z);
    printf("%i\n", z);
    printf("%f\n", z);
    printf("%x\n", z);
    printf("%o\n", z);
    printf("%%c\n", z);

    scanf("%c", &z);
    printf("%c\n", z);

    scanf("%s", &s);
    printf("%s\n", s);

    scanf("%x", &z);
    printf("%i\n", z);
    return 0;
}
```

Ausführung und Ein-/Ausgabe:

```
Beliebiger Text
<Eingabe: a>
<Eingabe: abc>
<Eingabe: a>
```

- Zeiger
  - Möglichkeiten, um eine Variable (analog: Funktion) anzusprechen:
    - über ihren Namen
    - über ihre Speicheradresse
  - Adressermittlung mit dem &-Operator
  - Deklaration und Dereferenzierung von Zeigervariablen mit dem *-Operator
- Adressarithmetik:
  - Ziel: "Rechnen" mit Adressen/Zeigern
  - Adressen: feste Länge (z.B. 32 Bits = 4 Bytes pro Adresse)
  - Länge der Daten an einer bestimmten Adresse abhängig vom Datentyp – zum Beispiel:
    - Integer: 4 Bytes
    - Character: 1 Byte

- Beispiel: Zeigervariablen
  - Was liefert dieses Programm (zeiger1.c)?

```
#include <stdio.h>

int main() {
    int *z1, *z2; //Zeigervariablen
    int n; //gewöhnliche Integer-Variable
    z1 = &n;
    z2 = z1;

    scanf("%i", &n); //liest Eingabe

    printf("Inhalt von n: %i\n", n);
    printf("Adresse von n: %x\n\n", &n);

    printf("Referenz in z1: %x\n", z1);
    printf("Adresse von z1: %x\n", &z1);
    printf("Wert der Referenz: %i\n\n", *z1);

    printf("Referenz in z2: %x\n", z2);
    printf("Adresse von z2: %x\n", &z2);
    printf("Wert der Referenz: %i\n\n", *z2);
    return 0;
}
```

Ausführung und Ein-/Ausgabe:

<Eingabe: 24>

Inhalt von n: 24
Adresse von n: bfa361dc

Referenz in z1:
Adresse von z1: bfa361e4
Wert der Referenz:

Referenz in z2:
Adresse von z2:
Wert der Referenz:

- Beispiel: Adressarithmetik
  - Was liefert dieses Programm (zeiger2.c)?

```
#include <stdio.h>

int main() {
    int *z, n = 100;
    char c = 'e';
    char *ptr = "123fgh";

    putc(n, stdout);
    putc(c, stdout);
    putc(*ptr, stdout);

    putc(n++, stdout);
    putc(++n, stdout);

    z = ptr;
    ptr++;
    putc(*ptr, stdout);

    putc(*z, stdout);
    z++;
    putc(*z, stdout);
    printf("\n");
    return 0;
}
```

Ausführung und Ausgabe:

- Beispiel: Parameterübergabe
  - Welche Ausgabe liefert dieses Programm (params.c)?

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

int main(int argc, char *argv[]) {

  printf("Es wurden %i Parameter übergeben.\n", argc-1);
  printf("Dieses Programm heißt: %s\n", argv[0]);
  printf("Der erste Parameter ist: %s\n", argv[1]);
  printf("Der zweite Parameter ist: %s\n", argv[2]);
  printf("Der dritte Parameter ist: %s\n", argv[3]);

  /* Annahme: Parameter argv[2] ist eine Integer-Zahl */
  int param2 = atoi(argv[2]); /* string --> int */
  printf("Das Doppelte von %i ist %i.\n", param2, 2*param2);

  return(0);
}
```

- Beispiel: Parameterübergabe (Forts.)
  - Test 1: `./params 1 2 3`

```
Es wurden 3 Parameter übergeben.
Dieses Programm heißt: ./params
Der erste Parameter ist: 1
Der zweite Parameter ist: 2
Der dritte Parameter ist: 3
Das Doppelte von 2 ist 4.
```

  - Test 2: `./params 1 a`

```
Es wurden 2 Parameter übergeben.
Dieses Programm heißt: ./params
Der erste Parameter ist:
Der zweite Parameter ist:
Der dritte Parameter ist:
Das Doppelte von
```

- Beispiel: Parameterübergabe (Forts.)
  - Test 3: `./params 1`

```
Es wurden 1 Parameter übergeben.
Dieses Programm heißt: ./params
Der erste Parameter ist: 1
Der zweite Parameter ist:
Der dritte Parameter ist: LESSKEY=/etc/lesskey.bin
Segmentation fault
```

  - Test 4: `./params ab c`

```
Es wurden       Parameter übergeben.
Dieses Programm heißt: ./params
Der erste Parameter ist:
Der zweite Parameter ist:
Der dritte Parameter ist:
Das Doppelte von
```

- Speicherallokation und -freigabe
  - Feste Speicherreservierung:
    - Speicherbedarf steht zur Übersetzungszeit fest
    - Typisch: Variablendefinitionen

```
int x; int a[20]; int *p; char *c;
```

  - Dynamische Speicherreservierung:
    - Speicherbedarf/Größe des Speicherbereichs wird zur Laufzeit ermittelt
    - Speicherallokation- und freigabe (!) durch den Programmierer

- Beispiel: Dynamische Speicherreservierung (und EXIT-Makros)
  - Was macht dieses Programm (speicher.c)?

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

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    int *pointer;

    printf("n = %i\n", n);
    pointer = malloc(sizeof(*pointer) * n);
    if (pointer == NULL) {
        fprintf(stderr, "Nicht genug Speicher!\n");
        exit(EXIT_FAILURE);
    }
    free(pointer);
    return EXIT_SUCCESS;
}
```

  - Testen des Programms:
    - `./speicher n`
    - für n := 1 (10, 1000000, 1000000000, ...)

Ein- und Ausgabe (E/A)

MNM TEAM

- Dateien (Files)
  - Datei: Konstrukt zur Definition logisch zusammenhängender Daten
  - Eigenschaften von Dateien:
    - Sequenzielle Speicherung
    - Persistenz (dauerhafte Speicherung)
  - Operationen auf Dateien:
    - Lesen (read)
    - Schreiben (write)
    - Ausführen (execute)
  - Arten von Dateien:
    - Ausführbare Dateien → Programme
    - Nicht-ausführbare Dateien → Text, Medien, ...
    - Verzeichnisse
    - Gerätedateien

Ein- und Ausgabe (E/A)

MNM TEAM

- Dateiberechtigungen
  - Ausgangspunkt:
    - Linux als Mehrbenutzer-Betriebssystem
    - Unterschiedliche Nutzer = unterschiedliche Datei-Zugriffsrechte
  - Benutzerklassen:
    - Datei-Eigentümer (User)
    - Benutzergruppe (Group)
    - Alle anderen (Others)
  - Rechte:
    - Lesen (symbolisch: r; numerisch: 4, binär: 100)
    - Schreiben (symbolisch: w, numerisch: 2, binär: 010)
    - Ausführen (symbolisch: x, numerisch: 1, binär: 001)
  - Beispiele:

```
rwxr-xr-x (755)
rw-rw-r-- (664)
rw-r----- (640)
```

Ein- und Ausgabe (E/A)

MNM TEAM

- Dateiberechtigungen ändern
  - Unix-Kommando chmod (siehe `man chmod`):

```
chmod g+w filename
chmod ug+rw,o+r filename
chmod a+x filename
chmod o-wx filename
chmod 664 filename
```

  - Gruppenzugehörigkeit ändern:

```
chgrp newgroup filename
```

  - Fragen:
    - Was passiert, wenn der Eigentümer einer Datei mit der Berechtigungssignatur `---rwxrw-` versucht, seine eigene Datei zu lesen?
    - Hängt das Ergebnis davon ab, ob sich der User in der Benutzergruppe der Datei befindet?

Ein- und Ausgabe (E/A)

MNM TEAM

- Übersicht: Unix-Kommandos im Zusammenhang mit Dateien und Verzeichnissen
  - `ls`: Verzeichnisinhalt auflisten
  - `cd`: Verzeichnis wechseln
  - `mkdir`: Neues Verzeichnis erstellen
  - `rmdir`: Verzeichnis löschen
  - `touch`: Neue Datei erstellen
  - `rm`: Datei löschen
  - `chmod`: Zugriffsrechte (Datei oder Verzeichnis) ändern
  - `chgrp`: Benutzergruppenzuordnung ändern
  - `chown`: Eigentümer ändern
  - `cp`: Datei kopieren
  - `mv`: Datei verschieben oder umbenennen
  - `grep`: Datei nach gegebener Zeichenkette durchsuchen
  - `diff`: Unterschiede zwischen Dateien anzeigen
  - `less`: Dateiinhalt (seitenweise) ausgeben
  - `split`: Datei aufteilen

Ein- und Ausgabe (E/A)

MNM TEAM

- Filedeskriptoren
  - Dateitabelle: vom Betriebssystem für jeden Prozess (!) verwaltete Tabelle aller geöffneten Dateien – ein Eintrag enthält:
    - I-Node-Nummer (identifiziert den physischen Speicherort aller Dateimeta- und -adressinformationen)
    - Position innerhalb der Datei
    - Modus (read, write, attach)
  - Filedeskriptor: Nummer (Index) des jeweiligen Dateitabelleneintrags (einfache Integer-Zahl)
  - Standard-Filedeskriptoren (Konstanten in unistd.h):
    - 0: Standardeingabe (`STDIN_FILENO`)
    - 1: Standardausgabe (`STDOUT_FILENO`)
    - 2: Standardfehlerausgabe (`STDERR_FILENO`)

Ein- und Ausgabe (E/A)

MNM TEAM

- Filedeskriptoren (Forts.)
  - Konstante `OPEN_MAX` in limits.h:
    - Maximale Anzahl geöffneter Filedeskriptoren für einen Prozess
    - Entspricht der minimalen Anzahl an Filedeskriptoren, die das Betriebssystem immer zur Verfügung stellen kann
- Funktionen, die mit Filedeskriptoren arbeiten
  - `open()`: Datei öffnen
  - `close()`: Filedeskriptor (und damit Datei) schließen
  - `write()`: In eine geöffnete Datei schreiben
  - `read()`: Aus einer geöffneten Datei lesen

- Beispiel: Filedeskriptoren und open( ) (createfile.c)

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
  const char *new_file;
  int fd; // Filedeskriptor

  if(argv[1] == NULL) {
      fprintf(stderr, "Usage: %s file to open\n", *argv);
      return EXIT_FAILURE;
  }
  new_file = argv[1];
  fd = open(new_file, O_WRONLY | O_EXCL | O_CREAT, 0644);
  printf("fd value is: %i\n", fd);
  if(fd == -1) {
      perror("Error opening file");
      return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Ein- und Ausgabe (E/A)

MNM TEAM

- Optionen der Funktion open( )
  - Bearbeitungsflags beim Öffnen einer Datei (schließen sich gegenseitig aus!):
    - O_RDONLY: nur Lesen
    - O_WRONLY: nur Schreiben
    - O_RDWR: Lesen und Schreiben
  - Zusatzflags (können mit Bearbeitungsflags kombiniert werden):
    - O_CREAT: Datei erstellen, falls nicht vorhanden (Zugriffsrechte als dritter Parameter)
    - O_APPEND: Aktuelle Position auf Dateiende setzen (Anhängen)
    - O_EXCL (in Kombination mit O_CREAT): Datei wird nur geöffnet, falls sie zuvor noch nicht existierte
    - Weitere: siehe `man 2 open`
  - Beispiel aus der vorherigen Folie:

```
open(new_file, O_WRONLY | O_EXCL | O_CREAT, 0644);
```

- Headerdateien
  - Einbindung mit der Präprozessoranweisung `#include`
    - `#include "/baum/ast/dateiname.h"`
    - `#include <dateiname.h>`
  - Vordefinierte Headerdateien (Definitionsdateien für Bibliotheksfunktionen):
    - erweitern den C-Befehlssatz
    - Einige wichtige:

      Standard-Ein-/Ausgabe: `stdio.h`

      Mathematische Funktionen: `math.h`

      Speicherverwaltung und Programmablaufsteuerung: `stdlib.h`

      Datum und Zeit: `time.h`

      Elementare E/A-Operationen auf Dateien: `fcntl.h`
    - Hilfe zu einer Definitionsdatei: `man dateiname.h`
  - Definition eigener Headerdateien möglich

- Make
  - Linux-Befehl `make`:
    - Prinzipiell: Ausführung beliebiger Aktionen
    - Wichtiger Einsatzbereich: Übersetzen von Programmen → mehrere C-Quellcodes zu einem ausführbaren Programm kompilieren
    - Dabei werden nur die tatsächlich erforderlichen Übersetzungen (entsprechend der definierten Abhängigkeiten) vorgenommen.
  - Makefile definiert hauptsächlich:
    - Ziele (targets)
    - Abhängigkeiten (dependencies)
    - Regeln (rules)
  - Aufruf: `make ziel`

- Beispiel: Übersetzen mit make
  - Gegeben: Zwei Programme aus drei Quellcode-Dateien
    - Programm 1: p1_modul1.c und p1_modul2.c
    - Programm 2: p2_main.c
  - sollen zu den ausführbaren Programmen myprog1 und myprog2 kompiliert werden
  - Mögliches Makefile:

```
# Makefile

CFLAGS = -g -Wall
CC = gcc

all: myprog1 myprog2

myprog1: p1_modul1.c p1_modul2.c
          $(CC) $(CFLAGS) -o myprog1 p1_modul1.c p1_modul2.c

myprog2: p2_main.c
          $(CC) $(CFLAGS) -o myprog2 p2_main.c

clean:
          rm –f *.o myprog1 myprog2
```

- Fragen zu diesem Beispiel
  - Was sind `CFLAGS` und `CC`?
  - Wozu wird `$` benötigt?
  - Welche Elemente sind Ziele?
  - Welche Elemente sind Abhängigkeiten?
  - Welche Elemente sind Regeln?
  - Was passiert, wenn zwei mal direkt hintereinander `make all` aufgerufen wird?

Zusammenfassung

MNM TEAM

- Grundlagen von Betriebssystemen
- Aufgaben und Konzepte der Systemprogrammierung
- Lizenzierung, Standardisierung, Entwicklung

- Erstes C-Programm: "Hello World"
- Übersetzung von C-Programmen
- Einfache Konsolen-E/A und Umwandlungszeichen
- Zeigervariablen und Adressarithmetik
- Parameterübergabe
- Dynamische Speicherreservierung
- (EXIT-)Makros
- Dateien, Berechtigungen und zugehörige Unix-Kommandos
- Filedeskriptoren und elementare E/A-Funktionen
- Headerdateien
- Makefiles